R94:17 ISSN 1100-3154

A Compact Intermediate Format for SIMICS

Peter Magnusson
psm@sics.se

David Samuelsson
davids@sics.se

September 1994

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

Abstract

Instruction set architecture (ISA) simulators are an increasingly popular class of tools for both research and commercial purposes. Common applications include trace generation, program development, and compatibility support. A major concern with ISA simulators is performance and memory overhead. A common technique for achieving good performance is to use threaded code, which involves translating the target object code to an intermediate format which is subsequently interpreted. We describe such an internal format, which we call the 64-bit format, that is compact and meets a range of requirements in terms of flexibility and simplicity. We show how a simulator using this format can be implemented efficiently by taking advantage of extensions to the C language supported by the GNU C compilers. We have used the format to write the core interpreter in SIMICS, a system level multiprocessor simulator that supports the Motorola 88110 and the Sparc v8 instruction sets.

Keywords: Intermediate representation. Interpreter. Simulator. Instruction set architecture. Sparc. m88110. C. GCC.

A Compact Intermediate Format for SIMICS

Peter Magnusson and David Samuelsson

Swedish Institute of Computer Science
Box 1263, Kista
Stockholm, Sweden
{psm,davids}@sics.se

September 1994

Abstract

Instruction set architecture (ISA) simulators are an increasingly popular class of tools for both research and commercial purposes. Common applications include trace generation, program development, and compatibility support. A major concern with ISA simulators is performance and memory overhead. A common technique for achieving good performance is to use threaded code, which involves translating the target object code to an intermediate format which is subsequently interpreted. We describe such an internal format, which we call the 64-bit format, that is compact and meets a range of requirements in terms of flexibility and simplicity. We show how a simulator using this format can be implemented efficiently by taking advantage of extensions to the C language supported by the GNU C compilers. We have used the format to write the core interpreter in SIMICS, a system level multiprocessor simulator that supports the Motorola 88110 and the Sparc v8 instruction sets.

KEYWORDS: intermediate representation, interpreter, simulator, instruction set architecture, sparc, m88110, C, GCC

1. Introduction

Instruction set architecture (ISA) simulators give excellent control over program execution.[1] This characteristic has made them popular among both hardware and software designers. Computer architects use ISA simulators for trace generation or on-the-fly gathering of statistics. Software developers are increasingly conscious of the benefits of simulators when faced with difficult problems, such as debugging or profiling software systems where standard methods are inappropriate.

Traditional use of simulation as an instrument for architecture studies has often suffered from the consequences of poor simulator design. If the simulator is slow or has a large memory overhead, then only small programs (“toy benchmarks”) can be studied. If the simulator fails to simulate system level effects, the resulting statistics will be non representative of real workloads. Among the

[1] An instruction set architecture (ISA) simulator simulates a computer at the instruction level. Instructions are interpreted one by one and their effects are allowed to update the simulated processor’s state. This can be done at several levels; SIMICS simulates an ideal processor model that executes one instruction per time unit (cycle).

more important system level effects that are commonly omitted are those caused by page faults, interrupt-driven I/O, cache interference, and multiprogramming. The common reason for their omission is that they are difficult to support, especially in fast simulation techniques such as variations of direct execution.

Another common problem is that simulators are seldom fully interactive. A fully interactive simulator responds quickly and has a pleasant front-end, but primarily is fully deterministic; regardless of what actions the user does (setting breakpoints, inspecting machine state, single-stepping) the execution of the program should not be affected.

At SICS we have developed a simulator—SIMICS—which demonstrates that these design difficulties can be dealt with. SIMICS is both efficient and multi-purpose:

- *computer architecture investigations;* a common domain for simulators, the purpose here is to understand the frequency and character of hardware events triggered by software, often related to the memory hierarchy.[2]
- *program profiling;* traditional techniques of detailed program behavior analysis are too invasive or inflexible for complex systems, such as real-time operating system kernels with extensive interaction among server components.
- *debugging*; simulators allow the debugging of code that is otherwise difficult to deal with, such as hand-coded system-level routines; furthermore, the unchallenged control over execution that a simulator can deliver offers the opportunity for new approaches to program debugging.

The current features of SIMICS includes:

- runs Sparc or m88110 binaries, either user level (supported with simple Unix emulation) or system level (allowing kernel binaries to run).
- supports full memory management unit semantics.
- simulates a uni- or multiprocessor.
- simulates data cache, including linking with memory hierarchy simulators written by a user.
- deals correctly with supervisor and user addresses, handling page faults etc.
- supports shared physical address space (e.g. bus-based multiprocessor), distributed (e.g. message-passing architecture), or hybrids.
- can profile memory usage (working set) as well as traditional code profiling.
- can have a symbolic front-end (GDB 4.11 or Xray)[3] or run in batch mode.
- is portable (currently runs on SunOS 4.1, Solaris, or HPUX).
- completely deterministic.
- is fast and has low memory overhead.

Most of the features can be selected interactively. Thus, SIMICS can simulate a four-processor architecture with a shared memory bus and 64K direct-mapped first level cache, or a 16-processor distributed memory architecture with message passing devices mapped into supervisor address space and 8K two-way associative cache, all without recompiling.

The performance goal for SIMICS was a slow-down of less than 20, meaning that simulated code should run at most 20 times slower than equivalent hardware would.[4] We achieved a slowdown of

[2] A memory hierarchy is a hiearchy of caches, possibly using different coherency schemes. In evaluating multiprocessor memory systems, it is often of interest to look at the frequency of different coherency protocol transactions.

[3] The GNU project debugger from FSF, and a debugger from MRI, respectively.

[4] In many circumstances the concept of equivalent hardware is vague. Our quoted performance is from the meaurement where the equivalent hardware was the host itself. Even so, slowdown varied (getting worse with progressively faster hosts).

13-17. It is important to keep this goal in mind; a saving of one host instruction on the average interpreted target instruction improves performance on the order of 5%.[5]

Our ability to develop SIMICS to support all this functionality and still remain flexible and efficient is very much due to a carefully designed intermediate format. The topic of this paper is a description of the intermediate format, a discussion of its benefits, and some notes on experiences in using it in a full implementation.

2. Previous Work

Interpreting an abstract machine was traditionally done in a single step (Knuth 73; Calingaert 79).[6] A more recent approach is to split the decode/simulate step into two phases, storing an internal code that is easier to interpret (Lang et al 86; Bedichek 90). In May's terminology, the former is a *first-generation simulator* and the latter a *second-generation simulator* (May 87).

There are principally two forms of second-generation simulators, those that interpret the intermediate format and those that execute it directly. We'll call the former *interpretive translation* and the latter *direct translation*, see figure 1.[7]

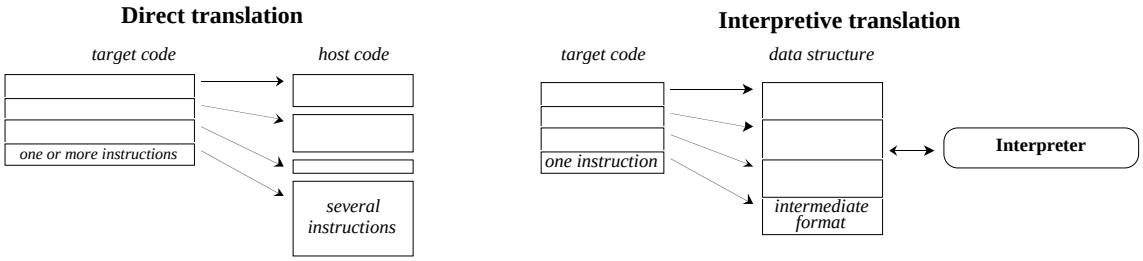

Figure 1 - Translation alternatives

Direct translators have become increasingly popular over the past two years. Commercial packages use the technique to support execution of user binaries, either as a commercial offering by itself or as migration support (SoftWindows, Apple System 7.5, Digital's migration tools, IBM's x86 emulator for the PowerPC). Research tools are also beginning to use this technique to speed up interpretation speed, notably Shade (Cmelik and Keppel 94) and Mint (Veenstra 94).

There are several problems with direct translation when used in simulators. Primarily, the lack of support in existing implementations for the system level or multiple processors indicate that these aspects are difficult to implement with this technique alone.[8] Another problem has been poor

[5] An instruction set interpreter deals with the peculiarities of two architectures, the one it is simulating and the one that the interpreter itself is running on. Throughout this paper, we will call the former the *target* and the latter the *host.*

[6] An instruction set simulator is really simulating an abstract machine, unless the simulator code is mechanically derived from a hardware description of the processor.

[7] Direct translators are often called *incremental compilers*. We avoid this term, however, because it implicitly confuses orthogonal issues. A direct translator always generates host code (which in turn might call routines that are statically compiled in the simulator). However, it might do so only when necessary, in which case it is also an incremental compiler. An interpretive translator might also generate host code at run time, in which case it, too, is an incremental compiler (SIMICS is an example of this). The term "compiler" is used since this class of interpreters sometimes apply traditional compiler techniques during the translation, such as peep-hole optimization and register allocation

[8] A system level simulator can execute system binaries, i.e. operating system code. This requires simulating virtual memory, asynchronous interrupts, exception handling, memory-mapped devices, supervisor instructions, protection, etc. A multiprocessor simulator in addition has to deal with separate *physical* address spaces, different devices for different processors, message-passing devices, interprocessor interrupts, and some manner of simulating concurrency.

instruction cache behavior, but this appears to be an issue only with commercial products that need to run on low-end platforms. Finally, the resulting simulator is not portable.

The principal alternative, interpretive translation, is preferably implemented using threaded code (Bell 73). Bell distinguished between traditional programming (hard code), simple interpreters (interpretive code), and threaded code; see figure 2. In threaded code, the central decode-and-dispatch loop is eliminated and inlined into the end of the service routine, which becomes responsible for determining which service routine to call next. The effect was to eliminate unnecessary procedure call and loop administration overhead. There are several variations of threaded code (Deware 75), Bells approach being that of *direct threading.* Direct threading stores the actual memory address of the service routine and jumps directly.

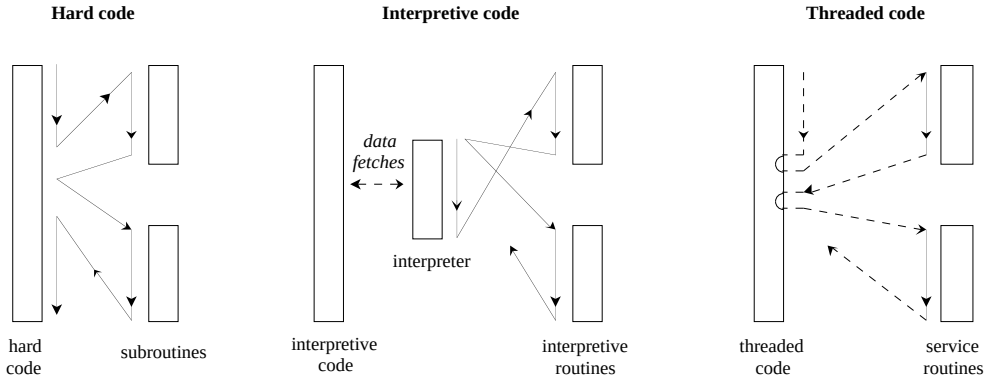

Figure 2: Threaded Code

Thus, direct translation simulate target instructions by always translating to host instructions and executing these. Interpretive translation, in contrast, translates the target code to an intermediate format that is faster to interpret. This subsequent interpretation preferably uses threaded code, and may or may not involve run time code generation.

3. Paper Overview

This paper is organized as follows. In section 4 we present the format itself. Section 5 discusses some benefits of using an intermediate code for ISA simulators; notably, we can play around with the mapping of target code to intermediate code.

Sections 6 through 8 then cover the benefits of the 64 bit format in particular. The first, section 6, describes the idea of inserted instructions, which allows intermediate code to be instrumented. Section 7 briefly mentions the importance of multiprocessor simulation issues in the choice of the 64 bit format. Finally, section 8 mentions the capacity of the 64 bit format to seamlessly support run time generated code.

Section 9 focuses on some implementation issues. We discuss the extensions to C supported by GCC that we found beneficial. Next we present a small, but complete, example. We finish the section by going into detail of how the full implementation of the interpreter core for SIMICS used the format.

We conclude the core of the paper with some performance figures in section 10.

4. The 64 Bit Intermediate Format

In this section we will describe the 64 bit intermediate format. We will also describe two alternatives as contrast.

The intermediate format needs to maintain enough information to carry out the instruction. Each entry needs to specify what piece of code should be called, and what parameters need to be passed.

It is tempting to do as much work as possible statically when generating the intermediate code. Consider a triadic register-to-register instruction like *add.* Increasing static work will resolve this instruction into four pointers: one for the simulator routine for the *add,* and three for the parameters. This means that seven memory accesses need to be performed to execute the instruction: one for the service routine pointer and two for each of the three registers (since they need to be de referenced). Since most instructions are simple to simulate, this means that memory bandwidth quickly limits simulator speed.

An example of such a "fully evaluated" format is the one used in g88, shown in figure 3 (Bedichek 90). This format also used a literal pool for 32-bit constants. This reduces the number of instructions that the interpreter needs to support; for example, there is no longer a need to distinguish between "add" with register and "add" with literal.

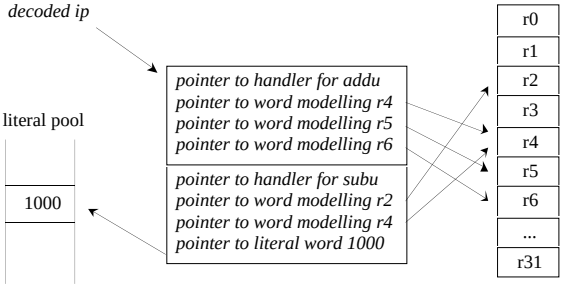

Figure 3: Intermediate format of g88

To reduce the required memory bandwidth, we make the following observation; the information kept in these pointers is highly redundant. They all point into a single, small array containing the contents of the simulated registers. Figure 4 illustrates how three seven bit values are stored as (scaled) offsets into an array representing the target processor state.

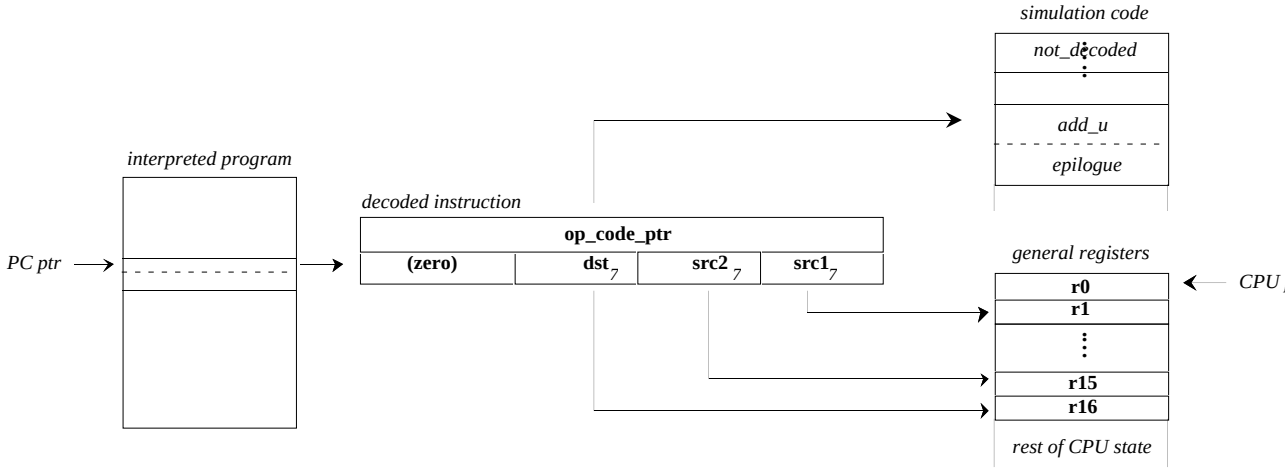

Figure 4- 64-bit intermediate format

To interpret this instruction, we now need four memory accesses to fetch five words. A single load double will fetch all 64 bits into a global register pair, rOP and rNEXT. The register offsets can be extracted from rOP and the register values fetched by indirecting over a pointer—*rCPU*—to the

CPU data structure. In the example in figure 4 we really only need five bits (since we have at most 32 registers), but we have room to store them pre-scaled.[9]

Considering that the entire routine will be on the order of fifteen instructions, the reduction in memory accesses is significant.

The 64-bit format has several benefits, which we list below. For each item, we also note the section that goes into detail.

1. the 64-bit entry can typically be read by a single "load double" instruction (section 9.4).
2. there is small data dependency on the fetching of the intermediate format; the pointer (rNIP) is known since entry to the routine, and rOP is not required until after the entry of the next. (section 9.4)
3. there is no restriction on the number of intermediate instruction types (section 5)
4. there is no restriction on memory location of service routines (section 8)
5. we can mix run time generated code with static service routines with no switching overhead (section 8)
6. the opcode does not need further processing to be used (section 9.2)
7. the format is independent of simulated processor (section 7)
8. it has enough room to encode the parameters efficiently (section 9.3)
9. there is now room for a full 32-bit pointer to a structure as a parameter, which is useful for pseudo instructions that require additional (perhaps a varying amount of) information (section 6)

We could take the compression further by observing that the *op_code_ptr* in figure 4 also contains redundant information; the bottom bits are zero, and the top bits are constant. By aligning the service routines on 128 byte boundaries, limiting the number of service routines to 512, and storing the top 16 bits of the first service routine in memory in a global register, we can get away with an intermediate format that is almost 32 bit, see figure 5. We say "almost" because not all instruction parameters will fit in the 23 bits left over (such as instructions with 16 bit immediates and two registers). Thus, we still require 64 bits of memory allocation, though most instructions will not fetch the second word. We call this format the compressed 64 bit format.

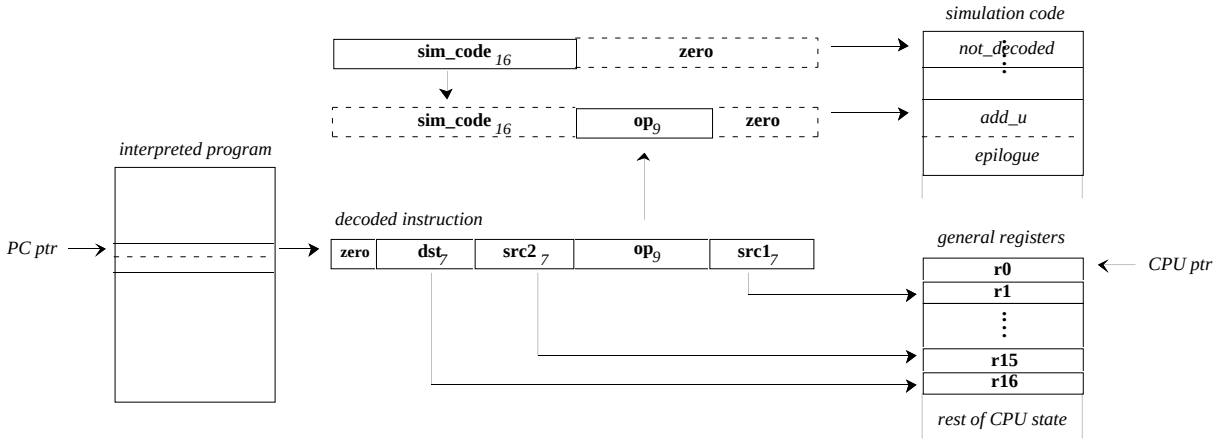

Figure 5: Compressed 64-bit format

One of the major problems with this compressed alternative is implementing the direct threading. Aligning service routines in memory is possible, but difficult in practice.[10] Referring back to the list

[9] Pre-scaling saves us some work on host processors that do not have bit field extraction instructions.
[10] And certainly not portable.

of benefits of the 64 bit format, we can observe that we lose items 3, 4, 5, 6, 8, and 9. Furthermore, items 1 and 2 mitigate, if not eliminate, any performance benefit. For these reasons, we do not believe that is worthwhile to compress the 64 bit format further.

5. Instruction Mapping

The very nature of an intermediate representation frees us from a 1:1 mapping of target instructions to service routines. We can make complex decisions during translation which will not impact performance upon execution.

5.1. Simulator Control Codes (SICC)

A simulator control code is a service routine that has no correspondence at all in the target object code. SICCs are used to signal internal aspects. An example of a SICC is the SIM_not_decoded and SIM_not_decoded_profile markers mentioned later in the paper.

5.2. Specialized Instructions

We can improve the performance by observing that some variations of instructions are very common. Below are some examples of common variations that we detect in SIMICS:

- return from subroutine. This is often a jump to a particular register, as defined by the programing standard for the particular processor.
- non-indexed memory accesses. Memory operations typically indirect over a sum, where the sum is either two registers or a register and a literal. A common operation is where one of the values is zero, making the operation a simple indirection.
- stack accesses. Memory operations often indirect over the stack pointer with a small literal offset (accesses to auto variables).
- small integer arithmetic. Add and subtract operations are very common, and they are completely dominated by small integers (typically low exponents of 2).
- on page branches. Since we are simulating an instruction MMU, we need to assert the legality of every (implicit) instruction fetch. However, static branches on the same page will never miss the instruction TLB since if they did, the branching instruction itself could not have executed.
- register manipulation. Several forms of register-to-register operations are very common, including copying a register, clearing a register, and loading a small literal.

As an example of how dominating these common cases are, table 1 lists the classification of instructions executed by the 023.eqntott SPECint92 program (taken over all 1.25 billion instructions). Despite the high figures, several of the above variants are not implemented.

023.eqntott	**Register manip.**	**Small literals**	**Branch on page**	**Other**
% Instructions	38.0%	9.9%	29.2%	22.9%

Table 1: Instruction Specializetion

5.3. Alternate Versions

Specialized versions of instructions really implement variations of N:1 translations. Any given instruction can only be translated to one instruction. It is also possible to do the reverse, i.e. a

particular instruction could be translated to different service routines depending on the circumstances.

An example of how we utilize this is with simulation of data caches. This requires every single memory access to check the state of the simulated cache. In SIMICS, for every memory instruction service routine there is a corresponding cache version. This way, cache simulation can be turned on and off interactively, with no need to recompile the simulator. When the "mode" changes, we discard previous translations and translate differently the next time the instructions are executed.

5.4. Multiple Simultaneous Translations

So far in this section we have assume that only a single translation is active at any give time. In fact, we can have multiple translations. What this means is that when an instruction is simulated in a particular mode, it is translated within that mode. When the mode changes, we do not have to throw away previous translations. Instead, we just change the appropriate pointers and continue interpretation of intermediate code on another set of pages.[11]

Currently we use this to support efficient simulation of global condition codes. On processors such as the Sparc, global condition codes are set by general instructions such as add or subtract. Conditional branch instructions use these codes to decide on whether to branch or not. A straight-forward implementation of global condition codes would be slow; four condition codes (the classic N,C,Z, and V) have to be laboriously generated even though all four are seldom used. Instead, we implement an "optimal" mode and a "non-optimal" mode. The former mode supports a subset of all possible condition codes and is faster. The latter mode is slow but complete. Whenever an instruction in optimal mode generates condition codes that cannot be represented, it switches to non-optimal and re-executes the (corresponding) service routine. In a similar fashion, we change back as soon as possible. For all the programs we have run, more than 99.9% of instructions execute in optimal mode.

6. Inserted Instructions

We mentioned earlier that the 64-bit format allows a full 32-bit pointer as a parameter. A useful example is code profiling. Profiling code counts the number of times a basic block has been executed. To achieve this, when we translate a basic block to intermediate code, we replace the first instruction with an "inserted instruction".

The idea is to allocate a data structure with information on the "real instruction". The inserted instruction then increments a counter and jumps straight to the service routine. Since all information in the 64 bit format is loaded into registers before the service routine is entered, it is not dependent on where the information came from.

On a Sparc host, an inserted profiling instruction that updates a counter and re-dispatches the "real" service routine takes six instructions. Note that this is recursive; several inserted instructions can be cascaded without being aware of each other.

The instruction profiling in SIMICS is implemented using inserted instructions.[12] It supports interactive activation and de-activation of profiling during a debugging session. It is efficient; the performance of SIMICS drops approximately 10% when profiling is activated.

---

[11] Currently in SIMICS, switching mode only requires changing the values of four gobal registers in addition to the mode descriptor (which is in memory).

[12] The real implementation is complicated by the need to track branches into basic blocks that have been profiled, branches into blocks that have not yet been translated, and the manipulation of profile entries when multiple translations are active. Also, system-level simulation adds further complications; a basic block may be interrupted by an exception and not re-entered, the program may generate code at runtime (such as trap vectors), etc. All this manipulation is supported by using inserted instructions and SICCs.

7. Multiprocessor Considerations

Since the decoded format is independent of the target processor, decoded instructions are identical for all processors in a multiprocessor simulation. This is beneficial, as it reduces the memory requirements for simulating large multiprocessors. Also, this means that processor switching can be done by changing the values of a few registers.

8. Partial Translation

We mentioned in the section on the 64-bit intermediate code that the service routine could be implemented at run-time. In this section will describe in a little more detail what we meant by this.

Very little of the work done by an interpreter is necessary. This is especially true when the host and target architecture's are similar. In the eight instructions needed for doing a simple triadic add, for example, only one instruction performs the actual add. It has long been known that this overhead can be reduced significantly by binary translation—selecting a segment of target code and generating corresponding code for the host architecture.

Traditional approaches have had several difficulties. Memory problems result from code expansion and large jump tables. Several features useful to researchers are difficult to support, such as simulating MMU, cache, asynchronous events, multiprocessors, profiling, and debugging. Finally, the resulting simulator is difficult to develop and port.

We improve on traditional approaches by integrating interpretation with run-time code generation. Combining interpretation with code generation is an old concept but generally avoided due to its complexity. However, our experience is that the combination of 64 bit intermediate format, direct threading, and GCC support for global register allocation greatly simplifies the design.

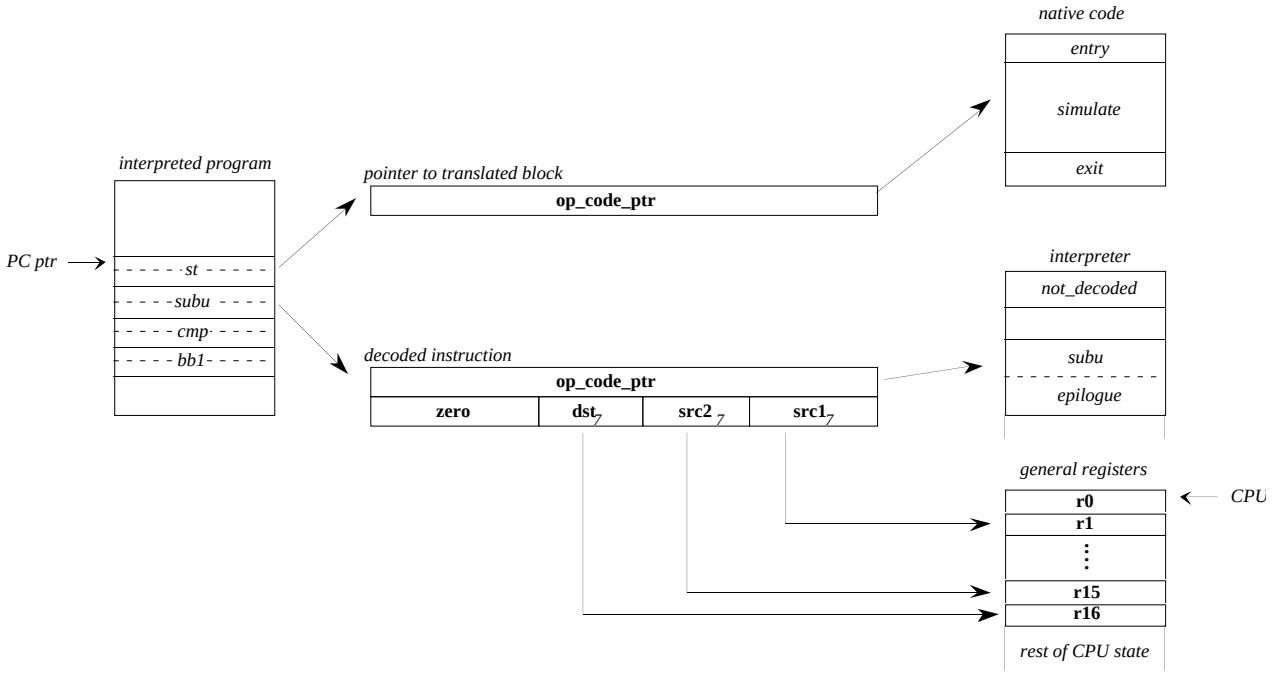

Figure 6 - Partial translation

The idea is to have an interpreter core that can handle any situation that arises. We then use the inserted instructions technique described earlier to detect sections of code that are executed often. These code sections are translated in a manner that can coexist fully with the interpreter. The code is generated on optimistic assumptions, including assertions to confirm during execution that the assumptions are valid for every repetition of the same code. Should any assumption fail, the

instructions revert to being interpreted on an instruction-by-instruction basis. Among the assumptions are: no events, no page faults, and no exceptions.

Consider figure 6. Four 88110 instructions constitute a small loop. After gathering statistics, the simulator determines that this inner loop is suitable for translation. All four are translated, and a pointer straight to the generated code is placed in the intermediate format. Any instruction dispatching the first instruction will jump straight to this code. No “mode switch” is needed. The interpreter routines and the generated code share a set of global registers to communicate program counters, event counter, processor structure pointer, etc. The remaining registers are used as scratch registers by the generated code to locally map target registers to host registers.[13]

Note that should any of the instructions in the loop other than the first be dispatched, the remaining instructions in the loop will be interpreted one-by-one, until the first one is dispatched, upon which the generated code will be re-entered.

The generated code shares the simulator with the interpreter. If an event occurs, or there is a page fault, the generated code will carefully “roll back” to a correct state and dispatch the relevant service routine. The generated code supports correct simulation of MMU and/or cache.

In SIMICS we have implemented a code generator for a Sparc host to prove that partial translation works. The generated code is very naïve—it is single pass, several obvious optimizations are possible, and only a handful of common instructions are translated. Despite this, the performance improvement is on the order of 30% for the programs that we have measured.

9. Some Comments on Implementation

In this section we discuss some implementation-specific aspects of the intermediate format. We being by mentioning some extensions to the C language, supported by GCC, that we take advantage of.[14] Next, we describe a simple but complete example. After that, we discuss some aspects of the full implementation.

9.1. The GNU C Compiler

GCC (Gnu C) is a generally available, portable C compiler (Stallman 92). GCC 2.x currently supports most major platforms from a single set of sources, and is the most innovative and useful broadly available C compiler. The C language was originally heralded as a structured language with a small semantic gap to the underlying hardware. This is no longer the case; several features of modern architectures are difficult or impossible to express in C. We have exploited several extensions to C that GCC supports, all of which directly contribute to a better simulator.

GCC 2.x supports label types for computed gotos. This allows pointers to code to be extracted and manipulated. Jumping to these computed labels is faster than a procedure call.[15] Here is an example of the syntax:

[13] Note that this works fine in combination with static code, since we leave and re-enter the static code on or just before labels. Since threaded code in SIMICS uses calculated gotos, GCC cannot know where we will go and hence will flag all scratch registers as dead except for the ones we have explicitly declared global.

[14] SIMICS will work without these extensions, but will be slower.

[15] The jump compiles to a jump on register value on the host. A procedure, by contrast, has entry and exit code associated with it.

```
        void * op_ptr;
        op_ptr = && label_A;
        goto *op_ptr;
        ....
label_A:
        /* the goto comes here */
        ....
```

The computed goto is efficient is difficult to use. We need to extract tables of labels for the rest of the simulator to use when manipulating the intermediate format. This leads to a software organization problem in keeping the various uses synchronized, since we're adding and subtracting service routines continuously. We solve this by using a single file to define the existence of different service routines, and using a script we generate header files from this.

Another problem is that the manipulation of these labels is sometimes incorrect. It is difficult to pinpoint these errors, because a debugger will not identify where the error occurred, only that we ended up trying to execute data somewhere.

One approach we have used to systematically root out bugs of this nature is validation of accessed values. GCC allows statements within an expression. The value of the last statement is returned as the value of the expression. For example, if we want to check that a particular value is always reasonable, we can assert the value before delivering it. Consider the following example:

```
#ifdef DEBUG
#define THE_VAR   ({if (the_real_var == 0) then error(); else the_real_var;})
#else
#define THE_VAR   the_real_var
#endif
```

In the example, THE_VAR is used to "indirect" over *the_real_var*. Thus, we do not lose performance unless we wish to debug, in which case we assert the sanity of *the_real_var* each time that it is accessed (in this case it must never be zero).

It is a common problem of interpreters that a small number of variables are accessed very often and in very many places. We make heavy use of hard register allocation inside SIMICS, something that GCC supports adequately.

Support of 64 bit integer data types is also very useful, and GCC does this with the "long long" data type.

9.2. A Small Example

In this section we show a small yet complete example of how to code our format using GCC extensions. The code below makes the necessary declarations and shows how to code a simple service routine.

```
struct processor {
  unsigned long regs[32];
};

/* the parameter field encoding for the add service routine */
struct format {
  unsigned dst:18;
  unsigned src1:7;
  unsigned src2:7;
};

register long long                    rTWO_PARAMS asm ("%g2");
```

```
register void *                 rNEXT asm ("%g2");
register struct format          rOP asm ("%g3");
register long long *            rXIP asm ("%g5");
register long long *            rNIP asm ("%g6");
register unsigned long          rEVENT asm ("%g1");
register struct processor *     rCPU asm ("%g7");

extern void handle_event();

#define REGS(field)             *(unsigned long *)((char *)rCPU + rOP.field)

#define epilogue()              \
do {                            \
  rNIP = rXIP;                  \
  rNIP ++;                      \
  rTWO_PARAMS = *rXIP;          \
  if (--rEVENT != 0) {          \
    goto * rNEXT;               \
  } else {                      \
    handle_event();             \
  }                             \
} while (0)

void add()
{
  REGS(dst) = REGS(src1) + REGS(src2);
  epilogue();
}
```

The add() function illustrates what a service routine might look like.[16] As described further on, the parameter macros would be more complex to support a full instruction set. The above code compiled for the sparc host results in fifteen Sparc instructions, which will execute in approximately twelve cycles on a SuperSparc processor.[17]

9.3. Encoding and Decoding of Instruction Parameters

The following is the “add” instruction service routine, as implemented in SIMICS:

```
L(add);
DST(ie,1,uint32) = SRC(ie,1,uint32) + SRC(ie,2,uint32);
epilogue();
```

The syntax is simple and intuitive, making it simple to implement a large number of service routines.[18] In this section we will describe the work done behind the scenes.

When SIMICS starts simulating a program, the object code of the program is transferred to memory.[19] When a page of object code is executed a corresponding page of intermediate code is allocated. When first allocated, a page of intermediate code is filled with a *SIM_not_decoded* SICC. *SIM_not_decoded* dispatches the routine that invokes the translator. Thus, execution flows from regular interpretation to decoding and back with little overhead.

---

[16] In a real simulator, a service routine would be a label within a large function, and the handle_event() function call instead be a “goto handle_event”.

[17] With possibly a one cycle penalty, depending on the cache configuration.

[18] Both the 88110 and Sparc support each require over 300 hand-coded service routines.

[19] If an operating system is being run, only the boot ROM is loaded, which in turn is responsible for loading the system binary from disk or over the network.

The translator reads the instruction from memory and generates the intermediate code according to the 64 bit intermediate format. The look-up function quickly finds a structure with information about the instruction to be decoded. The structure contains a pointer to the routine for interpreting the instruction and what format is used for source and intermediate encoding. The pointer to the service routine for the instruction will be stored as a second field of the intermediate code.

The add instruction in our example uses format “ie” (version “e” of format Roman numeral I). Below is a definition for its *MAKE* macro, followed by the respective extraction macros.

```
#define MAKE_ie(op, dst, src1, src2)   \
        {FIELD_1 = op; FIELD_2 = ((dst << 7 | src2) << 7 | src1) << 2; }
#define SRC_ie_1  (rCPU + (FIELD_2 & 0x7c))
#define SRC_ie_2  (rCPU + ((FIELD_2 >> 7) & 0x7c)
#define DST_ie_1  (rCPU + (FIELD_2 >> 14))
```

The *MAKE* macro saves the service routine pointer and packs the parameters. *FIELD_1* and *FIELD_2* identify the first and second word of the intermediate code, respectively. *rCPU* evaluates to the address of the simulated register file. Thus, the *SRC* and *DST* macros will evaluate to the addresses of the required registers. We are careful to place the bit vectors such that we minimize the number of shifts and masks required.

Now, let us return to the service routine for “add”, listed at the beginning of this subsection. The L() macro defines the entry point (typically a label). We will return to the epilogue macro in the next subsection.

The *SRC* and *DST* macros have a special syntax. We define three general macros for extracting data during execution. We use the macro concatenation feature of ANSI C to achieve the effect of an associative look-up (which is evaluated at compile time). These general macros are:

```
#define DST(format, field, type)  *(type *)DST_ ## format ## _ ## field
#define SRC(format, field, type)  *(type *)SRC_ ## format ## _ ## field
#define IMM(format, field, type) (type)  IMM_ ## format ## _ ## field
```

Taking *DST(ie,1,uint32)* from our example, it will first expand to:

```
*(uint32 *)DST_ie_1
```

and finally

```
*(uint32 *)(rCPU + (FIELD_2 >> 14))
```

which will correctly fetch the register value.

This layered implementation has several benefits. It isolates the writer of the service routines from the intermediate format, allowing this format to be varied and previous definitions to be re-used. It also allows the writing of the service routines in a syntax that is easier to visually inspect for errors. Finally, it facilitates automatic generation of service routines.

9.4. Epilogue

The epilogue of each service routine deals with dispatching the next instruction. Since this dispatch is very dependent on the way in which flow of control is managed, most of the discussion on this topic is beyond the scope of this paper. In this section we will briefly describe the simplest version as an illustration of how the service routine pointer is used, and why the 64 bit intermediate format is suitable for threaded code.

The simple epilogue eventually expands to something similar to:

```
rXIP = rNIP;
rNIP += 8;
if (--rEVENT == 0) {
  goto event_handler;
} else {
  rOP = *(u_long *) (rXIP +  4);
  dispatch(*(u_long *)rXIP);
}
```

The values *rXIP* and *rNIP* store the addresses of the intermediate code items that are in the instruction pipeline, thus supporting branch delay slots.[20] "8" is the size of the decoded instruction format. The "r" prefixes emphasize that most values are stored in registers.

Events are checked for on every cycle.[21] If no event occurs, rOP is loaded with the parameters for the next instruction. It is important to note that upon entry to an instruction, the relevant rOP has already been loaded. This means that code to fetch further parameters can execute immediately.

The *dispatch()* macro dispatches the next service routine, pointed to by rXIP, either by a direct jump (as explained in the section on the GCC compiler) or by using a switch statement.

It is worthwhile to code these portions by hand in assembler for every target we port to, since this single piece of code is executed several million times per second on high-end workstations. In the assembler implementation, rOP and the pointer to the next service routine are fetched with one load double instruction. Also, they are fetched early in the epilogue. With a Sparc host, the hand coded epilogue is six sparc instructions and is two cycles faster than the compiled C version.

10. Performance

The actual performance of SIMICS is difficult to quantify in any systematic manner. The simplest measurement of simulator performance in general is the number of instructions interpreted per second (measured in thousands, or kips). However, this number will vary greatly depending on the application and what features are enabled.

Table 2 lists three examples that illustrate the performance.[22] For each, we show the performance of different combinations of features. Data cache simulates a 16k, 2-way set associative first level cache with 32-byte cache lines. Instruction profiling counts exactly how many times an instruction in a particular memory location was successfully executed.

The first example runs a simple Sparc SunOS 4.1 user program, the infamous Dhrystone 2.1 benchmark (Weicker 84). In the measurement, it runs 100 000 iterations, which requires approximately 50 million instructions. Accurate data cache contents are maintained at a 2% performance loss. The second example runs a much larger Sparc program from the SPECint92 suite, which requires 1.25 billion instructions to complete.

Dhrystone 2.1	**No Data Cache**	**Data Cache**
No Instruction Profiling	2160	2117
Instruction Profiling	1824	1827

023.eqntott	**No Data Cache**	**Data Cache**
No Instruction Profiling	1717	1864

[20] A branch delay slot is when the effect of a branch is delayed one cycle. Both the 88110 and the Sparc processors have delayed branch instructions.

[21] Events are beyond the scope of this paper. They are used to simulate all asynchronous behavior such as switching processors or asking the front end if the user has pressed control-C.

[22] The measurements were done on a Sun SC2000.

Instruction Profiling	1564	1640

rt kernel (88k)	**No Data Cache**	**Data Cache**
No Instruction Profiling	496	478
Instruction Profiling	438	439

Table 2: Examples of SIMICS Performance

Our third example is considerably different from the other two. It is a measurement of the boot process of a commercial real-time kernel running on an 88110-based architecture. The absolute performance is much lower because the core interpreter is older and the boot-process is intensive in page faults, interrupts, and device programming. The cache performance is truly poor (>30% misses), since this version of the kernel did not cache portions of the address space.

Acknowledgements

Robert Bedichek's work has been the standard with which to compare, and he has always been helpful. In general, this work has profited from discussions with several people. In no particular order, we would like to thank Dave Keppel, Gordon Irlam, Bob Cmelik, Torbjörn Granlund, Per Berg, Staffan Skogby, Anders Landin, Seif Haridi, Björn Johansson, and Erik Hagersten.

Bibliography

Bedichek, R. 1990. "Some Efficient Architecture Simulation Techniques." In *USENIX - Winter '90*, 53-63.

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.

Cmelik, R. and Keppel, D. "Shade: A Fast Instruction-Set Simulator for Execution Profiling", in proceedings of *SIGMETRICS 1994.*

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

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.

Magnusson, P. 1992. "Efficient Simulation of Parallel Hardware." Masters thesis. Royal Instiute of Technology (KTH), Stockholm, Sweden.

May, C. 1987. "Mimic: a Fast System/370 Simulator." In *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques* (St. Paul, Minnesota). June 25-26, 1987, pp 1-13.

Motorola 1990, *MC88200 Cache/Memory Management Unit User's Manual,* Second Edition, Prentice Hall, 1990.

Motorola 1990, *MC88100 Risc Microprocessor User's Manual,* Second Edition, Prentice Hall, 1990.

Patterson, D. A. *Reduced Instruction Set Computers,* Communications of the ACM, Vol 28, No 1 (January 1985), pp 8-21.

Samuelsson, D. 1994. *System Level Simulation of the SPARC V8 Instruction Set,* SICS Technical Report, July 1994.

Sites, L.R., Chernoff A., Kirk M.B., Marks, P.M., and Robinson, S.G. *Binary Translation,* Communications of the ACM, February 1993, Vol. 36 No. 2, pp 69-81,

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

Sun 1990, *The Sparc Architecture Manual, Version 8,* Sun Microsystems, USA, December 1990.

Veenstra, J.E. and Fowler, R.J. "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors", proceedings of *MASCOTS 1994.*

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

Weicker, R. P. Dhrystone benchmark, C, version 2.1, Siemens AG, Postfach 3240, 8520 Erlangen, Germany.