Shade: A Fast Instruction-Set Simulator for Execution Profiling

Robert F. Cmelik
Sun Microsystems Laboratories, Inc.
2550 Garcia Avenue
Mountain View, CA 94043

David Keppel †
Department of Computer Science and Engineering
University of Washington, FR-35
Seattle, WA 98195

Technical Report UWCSE 93-06-06

ABSTRACT

Shade is an instruction-set simulator and custom trace generator. Application programs are executed and traced under the control of a user-supplied trace analyzer. To reduce communication costs, Shade and the analyzer are run in the same address space. To further improve performance, code which simulates and traces the application is dynamically generated and cached for reuse. Current implementations run on SPARC systems and, to varying degrees, simulate the SPARC (Versions 8 and 9) and MIPS I instruction sets. This paper describes the capabilities, design, implementation, and performance of Shade, and discusses instruction set emulation in general.

Shade improves on its predecessors by providing their various tracing capabilities together in a single tool. Shade is also fast: Running on a SPARC and simulating a SPARC, SPEC 89 benchmarks run about 2.3 times slower for floating-point programs and 6.2 times slower for integer programs. Saving trace data costs more, but Shade provides fine control over tracing, so users pay a collection overhead only for data they actually need. Shade is also extensible so that analyzers can examine arbitrary target state and thus collect special information that Shade does not "know" how to collect.

1. Introduction

Shade simulates (emulates) a processor, and is used to execute and trace programs. Today, Shade runs on SPARC [SPARC8] systems to trace SPEC [SPEC] benchmarks to help build better SPARC hardware and software. Shade cannot yet run operating system code nor multiprocessor applications.

Users write programs, called *analyzers*, which call Shade functions to simulate and trace *applications*, usually benchmarks. The most popular analyzers simulate caches and microprocessor pipelines.

Tracing in Shade is programmable and permits per-instruction access to the state (registers and memory) of the simulated program. Trace information flows from Shade to the analyzer through memory instead of pipes or files. Shade generates traces quickly and the level of tracing detail can be changed while the application is being simulated. Thus, Shade's traces can be recreated on demand, avoiding the processing and storage costs of conventional traces. When conventional traces are desired, it is straightforward to write Shade "analyzers" that generate traces in arbitrary formats.

The system where Shade and the analyzer run is the *host system*. The system where the application runs is the *target system*. The target system need not exist; through simulation, Shade offers applications a *virtual target*

---

† Academic consultant, Sun Microsystems Laboratories, Inc.

system (*virtual machine*). The virtual target may be an incomplete model of the target; for example, Shade does not yet simulate privileged instructions.

Shade uses dynamic compilation to increase simulation performance. At run time, application code fragments are compiled into special-purpose host machine code fragments called *translations*. Translations are executed directly on the host system to simulate and trace the application code. Translations are cached for later reuse to amortize compilation costs.

The translation that Shade compiles for an application instruction is essentially an optimized, in-line expansion of the code a conventional simulator would execute to simulate and trace the same instruction. Some of the work that a conventional simulator must do each time an instruction is simulated, such as fetching and decoding the application instruction, is instead done once by Shade when the translation is compiled. Compiling application instructions in groups provides additional optimization opportunities. For example, virtual application state that would normally be stored and loaded between simulating two application instructions can instead remain in host scratch registers.

Often, the host system is used directly to minimize simulation costs. For example, a target add instruction that sets condition codes can often be simulated with a host instruction that does just the same. This is more efficient than first performing an add, then computing the individual target condition code values based on the target operation, operands and results.

This paper is mostly about the design, implementation, and performance of Shade. The use of Shade is described elsewhere [Cmelik93b, Cmelik93c]. Unless otherwise noted, we are describing the version of Shade which runs Version 8 SPARC applications on Version 8 SPARC hosts.

2. User Interface

This section gives a brief tour of the Shade user interface, as seen by the Shade analyzer writer.

2.1. Loading

The function `shade_load` loads a new application program for simulation and tracing. The arguments to the function are the application program file name, command line argument list, and environment variable list. `shade_load` reads the text and data segments from the program file, allocates and clears a BSS (uninitialized data) segment, and allocates a stack, which is initialized from the command line arguments and environment.

Once the application is loaded, the analyzer is allowed to make certain system calls in the context of the application program. For example, the analyzer can make calls that redirect the application's I/O.

Analyzers can (sequentially) simulate multiple applications. This allows analyzers to perform the otherwise painstaking process of summarizing results for multiple-command benchmarks.

2.2. Simulating and Tracing

The function `shade_run` is called with an array of trace records and the array size. `shade_run` simulates the application while filling in trace records. It returns the number of instructions traced, which may be less than the number of instructions simulated. Zero is returned when the application has terminated.

Shade provides analyzers with control over tracing, so analyzers only pay for the trace data they want. Shade can directly record the following trace information:

- instruction address
- instruction text
- decoded opcode value
- effective address: data address for loads and stores, target address for control transfer instructions
- branch taken flag
- annulled (squashed) instruction flag
- values of integer and floating point registers used in the instruction; source register values are recorded before instruction simulation, destination registers after

The analyzer specifies which of these values should be recorded for each opcode. For example, an analyzer that simulates instruction and data caches can direct Shade to record instruction addresses and the annulled instruction flag for all instructions, plus effective memory addresses for loads and stores.

Shade knows how to gather the trace information listed above. Analyzers can gather additional information, such as special purpose register values, by supplying functions that are called just before or just after an application instruction is simulated. Different trace functions may be specified for different opcodes. Trace functions are not called for annulled instructions.

Each trace function is called with two arguments. The first is a pointer to the instruction's trace record, which may be used to store information collected by the trace function. The same trace record also serves to provide the trace function with the standard trace information described above. The second argument to the trace function is a pointer to the application's current virtual state structure, which contains the application's current virtual register values. The trace function may also access the application's "virtual" memory.

User trace functions can amend trace records but can neither create nor destroy them. User trace functions can alter the application's state, but doing so is neither recommended nor reliable.

Analyzers can enable and disable tracing according to instruction address.

The trace control configuration may be changed when Shade is not running the application (i.e. outside `shade_run`). This level of control is enough to allow long applications to be sampled, or completely analyzed in stages, with separate stages run in parallel by separate Shade invocations to reduce run time.

Sometimes it is desirable to regain control from `shade_run` before it has filled the trace buffer. For example, the user may wish for `shade_run` to return immediately before or after running an application system call. Yet if tracing is minimal, the entire application may be run in a single `shade_run` call. An analyzer can enable tracing[1] for all instructions, and set the trace buffer size to one instruction. Control then returns to the analyzer after running each application instruction. This is not very efficient, though, due to the overhead of switching between the analyzer and the application.

Conventional debuggers suggest several control mechanisms, including *instruction breakpoints* which would cause `shade_run` to stop before or after application instructions by opcode or address, and *data breakpoints* which would cause `shade_run` to stop before or after loading or storing data at user specified addresses. More generally, the user could supply functions which would be called during simulation, and, based on the current state of the application, decide whether `shade_run` should stop. None of these mechanisms are currently implemented in Shade.[2]

It would also be useful to allow *user simulation* functions, which would let the user modify or extend the base instruction set provided by Shade. No such facility is implemented in Shade.

3. Implementation

This section describes some of Shade's implementation details, especially those related to simulating and tracing application programs.

3.1. Simulating and Tracing

The function `shade_run` is responsible for simulating and tracing application code. At its heart is a small main loop (see Figure 1) that finds and calls a translation corresponding to the current target instruction. The translation is built by cross-compiling the target instruction into host machine code. The translation simulates the target instruction, updates the target PC, optionally records trace information, and returns to the main loop. In practice, each translation simulates several target instructions before returning to the main loop.

The main loop, translations, and most utility functions which are called by the translations, all share a common register window and stack frame. Several host registers are reserved for special purposes:

---

1. Note that tracing can be "enabled" even if no trace data is actually recorded in the trace buffer entries.

2. A Shade analyzer based on the GNU debugger `gdb` [Stallman87] has been developed for Version 9 SPARC software development [Evans92].

```
    initialize registers
continue1:
    sll     %vpc,..., %l0       ! hash application pc ...
    srl     %l0, ..., %l0       ! ...
    sll     %l0, ..., %l0       ! ... into tlb offset
    ldd     [%tlb + %l0], %l0   ! tlb lookup (loads %l0 and %l1)
    cmp     %vpc, %l0           ! tlb hit?
    be,a    run
    mov     %l1, %o0            ! translation address
    clr     %o7                 ! no chaining
continue2:
    save some registers
    mov     %o7, %o1            ! chaining address
    call    shade_trans         ! returns translation address
    mov     %vpc, %o0
    restore saved registers
run:
    check for signals
    jmpl    %o0, %g0            ! jump to translation
    nop
eot:
    save some registers and return
```

Figure 1. Simulator main loop

- pointer to application's virtual state structure (referred to in code examples as `vs`)
- base address of application's memory (`vmem`)
- application's virtual program counter; this is part of the virtual state, but is used enough to warrant its own host register (`vpc`)
- pointer to current trace buffer entry (`tr`)
- number of unused trace buffer entries (`ntr`)
- pointer to TLB (`tlb`)

Shade locates translations using a table called the Translation Lookaside Buffer (TLB). The TLB associates application instruction addresses with corresponding translation addresses. The main loop does a fast, partial TLB lookup. If that fails, the function `shade_trans` performs a slower, full TLB lookup, and if that fails, compiles a new translation and updates the TLB.

Shade then checks for pending signals that need to be delivered to the application. Depending on how the application wishes to handle the signal, Shade may terminate the application at this point, or arrange for invocation of an application signal handler. In the latter case, Shade continues simulating and tracing, but now in the application's signal handler. This process is repeated recursively if another signal arrives while in the application signal handler.

Translations do not return in normal fashion. Instead, they use `call` instructions to return to one of three points in or about the main loop: `eot`, `continue1`, or `continue2`.[3] Translations transfer control to `eot` when the trace buffer space is exhausted. Translations transfer control to `continue1` or `continue2` to pass control to the next translation via the main loop. The two return points are used by the translation compiler to implement an optimization called translation chaining.

3. Control transfer is similar to a continuation-passing style, but continuations are compiled into each translation.

3.1.1. Translation Chaining

Often, the execution of one translation always follows that of another. The two translations, predecessor and successor, can be directly connected (chained) to save a pass through the main simulator loop.

When the successor translation is generated first, the predecessor translation is compiled with a branch to the successor. Otherwise the predecessor translation is compiled with a `call` instruction to transfer control to `continue2` in the main loop. The `call` instruction saves its own address, which is later passed to `shade_trans`. `shade_trans` generates the successor translation, then overwrites the predecessor's `call` with a branch to the successor, thus chaining the two translations.

Translations for conditional branches are constructed with two separate exits instead of a single common exit, so that both legs may be chained.

Translations for register indirect jumps and software traps (which might cause a control transfer) cannot be chained since the successor translation may vary from one call to the next. For these translations, control transfers to `continue1` in the main loop.

Using the `continue1` return point is more efficient if the next translation has already been generated and appears in the TLB. Using `continue2` is more efficient when the partial TLB lookup following `continue1` would fail and control would fall through to `continue2` anyway.

3.1.2. Translations

Application instructions are typically translated in chunks which extend from the current instruction through the next control transfer instruction and accompanying delay slot. Translation also stops at tricky instructions such as software trap and memory synchronization instructions.

Shade arbitrarily limits the number of application instructions per translation (to 1024) in order to simplify storage allocation. The user's trace buffer size also limits translation size. Therefore, a translation may represent more or less than one basic block of application code, and one fragment of application code may be simultaneously represented by more than one translation. The TLB effectiveness is reduced when there are fewer application instructions per translation, or when there are multiple translations for the same application instruction.

Each translation may be broken down into three sections: a prologue, a body with a fragment for each application instruction, and an epilogue.

3.1.3. Translation Prologue

The translation prologue (see Figure 2) allocates trace buffer space for the translation. If there is not enough space, the full-buffer translation return is taken. Prologues are generated only for translations that trace instructions.

Analyzers control the tracing of annulled instructions. When annulled instructions are untraced, the required amount of trace buffer space is not known until the translation is run. For these translations, the prologue allocates enough space to trace the potentially annulled instruction, and the translation body makes the appropriate adjustment if the instruction is annulled.

```
prologue:
    subcc   %ntr, count, %ntr
    bgeu,a  body
    nop             ! hoist (replace) from body
    call    eot     ! full-buffer return
    add     %ntr, count, %ntr
body:
```

Figure 2. Translation prologue

The trace space requirements for each translation might instead have been saved in a data structure, and tested by the main loop. This would save the code space now used for translation prologues, but would require executing additional instructions to address and load *count*, and would be inconsistent with translation chaining in which translations branch to each other, circumventing the main simulator loop.

Here and elsewhere, the translation compiler *hoists* the target of a branch or call instruction into the delay slot to replace the `nop` instruction which was originally generated for lack of anything better. The branch or call instruction is then updated to target the instruction following the original target. Control transfer instructions are not hoisted.

3.1.4. Translation Body

The translation body contains code to simulate and optionally trace each application instruction. Simulation consists of updating the virtual state (registers plus memory) of the application program. Tracing consists of filling in the current trace buffer entry and advancing to the next.

3.1.4.1. Simulation

Figure 2a shows a sample application instruction, and Figure 2b shows the code generated to simulate it.

```
add     %r1, %r2, %r3
```

Figure 2a. Sample application code

```
ld      [%vs + vs_r1], %l1
ld      [%vs + vs_r2], %l2
add     %l1, %l2, %l3
st      %l3, [%vs + vs_r3]
inc     4, %vpc
```

Figure 2b. Translation body (no tracing)

The translation body first loads the contents of application registers `r1` and `r2` from the application's virtual state structure into host scratch registers `l1` and `l2`. Next, the translation performs the add operation. Then, the translation writes the result in host scratch register `l3` back to the virtual state structure location for application register `r3`. Finally, the translation updates the application's virtual PC.

The code that is generated to actually perform the application operation is very often one and the same instruction, but with different register numbers. Where the host machine is a poor match to the virtual target machine, or where we wish to virtualize the target machine operations, several instructions, or even a call to a simulation function may be used. At the other extreme, no instructions need be generated to simulate useless application instructions (e.g. `nop`).[4]

Application memory addresses must be translated for loads, stores, and other instructions which access memory. Translation is performed by adding the application memory address to the offset *base address*, stored in host register `%vmem`. The same offset applies to all application memory address translations.

Note that SPARC supports both register+register and register+immediate addressing modes for memory operations. Often (roughly 30% of the time) one of the target instruction's address operands is zero, and can be replaced by `%vmem`, which saves an `add` instruction.

Conceptually, the virtual PC is updated for each application instruction as shown here. In practice, it is only updated in the translation epilogue, or as needed in the translation body for tracing application instruction addresses.

A *Delayed Control Transfer Instruction* (DCTI) is a control transfer instruction such as a branch, jump, or call instruction, that takes effect only after the dynamically following instruction (which is called the *delay slot*) has been executed. In principle, Shade must maintain both a virtual PC and a virtual NPC (next PC) in order to simulate DCTIs. In practice, Shade is usually able to translate and simulate DCTIs in tandem with the statically following in-

4. Shade assumes application `nop`s serve a useful purpose; they are elided only when other instructions, such as simulation or tracing code, are available to replace them.

struction, and the NPC is maintained implicitly as PC+4. When the delay slot is itself a DCTI, or when simulation must be suspended between the DCTI and its delay slot, Shade must simulate the NPC in order to correctly follow the control flow. In these cases, Shade generates the less efficient Shadow translations, which are described below.

3.1.4.2. Tracing

Figure 2c shows the code generated to simulate and partially trace the sample application code. Note that tracing requirements may vary from one application instruction to the next. The tracing code that is generated varies accordingly.

```
st      %vpc, [%tr + tr_pc]  ! trace instruction address
set     0x86004002, %o0
st      %o0, [%tr + tr_iw]   ! trace instruction text
ld      [%vs + vs_r1], %l1
ld      [%vs + vs_r2], %l2
st      %l1, [%tr + tr_rs1]  ! trace 1st source register
st      %l2, [%tr + tr_rs2]  ! trace 2nd source register

mov     %vs, %o1
call    pre-instruction trace function
mov     %tr, %o0

add     %l1, %l2, %l3
st      %l3, [%vs + vs_r3]
st      %l3, [%tr + tr_rd]   ! trace destination register

mov     %vs, %o1
call    post-instruction trace function
mov     %tr, %o0

inc     4, %vpc
inc     trsize, %tr             ! advance in trace buffer
```

Figure 2c. Translation body (some tracing)

Instruction trace records have two parts. In the first part, Shade stores trace information it knows how to collect directly. This information is at fixed offsets (`tr_pc`, `tr_rs1`, etc.) from the beginning of the trace record. Shade stores some adjacent components with a doubleword store instruction, so trace records must be doubleword aligned.

User trace functions may record information in the second part of the trace record. The analyzer specifies an overall (first plus second part) trace record size (*trsize*), which is used to advance the trace buffer pointer.

One feature/kludge is that the first part of the trace record only needs to be long enough to hold the trace information that the analyzer wants Shade to record directly. With this in mind, more popular trace information (like instruction address and text) is located nearer the beginning of the trace record. A more general approach would allow the user to specify the trace component offsets within the trace structure. This approach would complicate the user interface but have negligible impact on compilation and execution.

The trace buffer entry counter (`ntr`) is decremented once, in the epilogue. Though an increment for the trace buffer pointer (`tr`) is shown here, it too is generally updated once in the epilogue. The host SPARC store instructions have a 13-bit indexing offset, so $\lfloor 2^{12}/trsize \rfloor$ trace records are immediately accessible (for simplicity, Shade uses only positive indexing offsets). When greater range is required, the trace buffer pointer is updated by the body as needed.

When a user trace function is called, live application state is first returned to the virtual state structure for use by the trace function. An exception is the virtual PC, which is available in the trace buffer entry.

3.1.5. Translation Epilogue

The translation epilogue (see Figure 3) updates the virtual state structure. The epilogue saves host registers that hold modified virtual register values. If the virtual condition codes have been modified, they too must be saved unless, as discussed later, the save can be avoided. The epilogue also updates the trace buffer registers `%tr` and `%ntr` if necessary.

Finally, the epilogue returns control to the main loop, or, if the translation is chained, branches directly to the translation which corresponds to the updated virtual PC value. The virtual PC remains in a host scratch register across translation calls. Upon return from a translation, it contains the address of the next application instruction to be executed.

```
epilogue:
     update virtual state structure
     update virtual PC

     inc      count * trsize, %tr
     call     continue1 or continue2   ! return to main loop
     nop
```

Figure 3. Translation epilogue

The `nop` at the end of the epilogue may be replaced by hoisting when the translation is chained.

3.2. Shadow Translations

Shadow translations[5] are generated instead of ordinary translations under the following circumstances:

1. to handle DCTI couples
2. to handle any DCTI when the analyzer has limited the trace buffer size to a single entry
3. to permit synchronous signal handling when an application instruction might trap, and trapping should cause the application's signal handler to be simulated "immediately"

Shadow translations maintain a virtual NPC and an annul-pending flag in addition to a virtual PC. The virtual PC value is set from the virtual NPC value. The new virtual NPC value is set to PC+4 for instructions which don't cause a control transfer; to the target address for `call`, `jmpl`, and taken branches; or to the fall-through address for untaken branches. The annul-pending flag is set when the next instruction should be annulled instead of executed.

The SPARC architecture specifies that executing a conditional branch with a DCTI in its delay slot has undefined behavior. For these illegal DCTI couples, Shade silently mimics the "obvious" behavior used in most of today's SPARC implementations.

Since Shadow translations may have different successors from one invocation to the next, they cannot be chained. Ordinary translations, though, can be chained to Shadow translations. Likewise, modified virtual condition codes must always be saved in Shadow translation epilogues because Shade cannot determine whether the application will set them before using them.

3.3. Translation Cache (TC)

The TC is the memory where translations are stored. In an earlier version of Shade the TC was organized as an array of fixed sized entries, with one entry per translation and one translation per application instruction. The application instruction address mapped directly to the corresponding TC entry.

The old organization was abandoned to allow multiple application instructions, variably traced, per translation. Now translations are simply placed in memory one after the other, irrespective of length, and the TLB associates the application instruction address with the corresponding translation address.

---

5. So called because they resemble the code used in Shadow, Shade's predecessor.

When more TC space is needed than is available, all entries in the TC are freed and the whole TLB is cleared. Since this deletes useful translations, the TC is made large so that freeing is rare. Translation chaining makes other freeing strategies tedious.

The TC and TLB are also flushed when there is any change in the tracing strategy, since tracing is hardcoded into the translations. This is not expected to happen often.[6]

If the application uses self-modifying code, the TC and TLB entries for modified application code will be invalid after the modification. SPARC systems provide the `flush` instruction to identify code that has been modified; many other systems provide equivalent primitives [Keppel91]. When an application flushes instructions, Shade makes the corresponding translations inaccessible. Note that translations may be reached either through the TLB or through translation chaining. If a flushed application instruction is later executed, a new translation for it is compiled.

To simulate the first `flush` instruction, Shade frees the TC, flushes the TLB, and disables future translation chaining. To simulate subsequent `flush` instructions, Shade flushes TLB entries which point to translations for the flushed instructions. Since Shade does not maintain enough information to tell exactly which translations cover which application instructions, TLB flushing is done conservatively, and some TLB entries will be needlessly flushed.

Older SPARC systems with unified instruction and data caches will correctly execute self-modifying code without the need for `flush` instructions. So applications exist (e.g. the dynamic linker) which don't `flush` when they modify code. By default, Shade makes TLB entries and chains translations only for application code inside the (read-only) text segment. As a result, instructions such as the dynamic linker (loaded at run time) and the code it loads are recompiled each time they are simulated. A Shade run-time option (`-flushbench`) may be specified to indicate that the application executes `flush` instructions where it should.

There are two Shade run-time options which control when Shade itself uses `flush` instructions. By default, Shade flushes all code that it generates or modifies. The first option (`-noflush`) tells Shade it need not execute `flush` instructions. The second option (`-flushp`) tells Shade to execute a single `flush` instruction between generating/modifying and executing an arbitrary amount of code.

Full flushing is required, e.g., for MicroSPARC which has split instruction and data caches and requires that software maintain consistency. Full flushing is also required when running Shade on itself. Partial flushing may be used, e.g., for SuperSPARC, which has consistent instruction and data caches, but which doesn't check instructions currently being executed against recently stored data.

3.4. Translation Lookaside Buffer (TLB)

The TLB is an array of lists of address pairs. Each pair associates an application instruction address with the corresponding translation address.

To find a translation, Shade hashes the application instruction address to produce a TLB array index. This TLB entry (address pair list) is then searched for the given application address.[7] If the search succeeds, the list is reorganized (if necessary) so that the most recently accessed address pair is at the head of the list. If the search fails, a translation is generated, and a new address pair is placed at the head of the list.

Lists are actually implemented as fixed length arrays, which makes the TLB simply a two-dimensional array of address pairs. The TLB may also be thought of as N-way set associative, where N is the list length. Since address pair lists are of fixed length, it is possible for address pairs to be pushed off the end of a list and lost, which makes the translation inaccessible via the TLB. The TLB is large enough that this is not a problem. The translation will likely also still be accessible via chaining from other translations.

3.5. Optimizing Translations

Shade uses an ad hoc code generator which generates code in roughly one pass. Some minor backpatching is later performed to chain translations and replace `nop`s in delay slots. The resulting code could no doubt be improved,

---

6. Retranslation overhead due to trace control changes could be reduced by caching the translation state (TC, TLB, etc.) by trace control strategy.

7. Recall that the main simulator loop only checks the first member of this list.

but the time spent in the analyzer usually dwarfs the time spent in code generation, simulation, and tracing combined. A run-time code optimizer seems unjustified in this case; the cost and potential bugs of a run-time code optimizer should be seriously considered as well.

Optimization is more important when Shade is used for pure simulation with no tracing or analysis. A different tradeoff might be made between the cost of code optimization and the quality of optimized code.

3.6. Conflicts of Interest

The decision to simulate, trace, and analyze all in the same process leads to conflicts over the use of per-process state and resources. Conflicts arise between the application program (i.e. the code generated by Shade to simulate the application), the analyzer, and Shade (translation compiler, etc.). Conflicts are resolved in one or more of the following ways:

1. Partition the resource (e.g. memory, file descriptors, signals). Part of the resource belongs to one contender, and part to another. If necessary, interfaces are provided to hide the partitioning, and perhaps provide some protection.
2. Time multiplex the resource (e.g. special purpose registers). Part of the time the resource belongs to one contender and part of the time it belongs to another. Additional processing time and storage are required for swapping.
3. Simulate (virtualize) the resource, invariably on behalf of the application (e.g. register windows). This is the least efficient approach.
4. Unprotected sharing (e.g. current working directory). Changes made by either effect both. This is generally a bad thing and the result of an incomplete implementation.

The following sections describe these approaches in more detail.

3.6.1. Memory

The virtual address space is partitioned. It is assumed that both the application and analyzer are free of bugs that would cause untoward memory references.

The analyzer, including the Shade run-time routines, runs as an ordinary program in the lower end of memory. `shade_load` places the application elsewhere in memory. The kernel picks the actual location for the application given a maximum size estimate provided by `shade_load`.

The application stack is placed near the text segment, just below what would be application virtual memory address 0. Thus, application stacks appear to the application as if they are located at the high end of memory. Application stacks do not enjoy the automatic extension provided by UNIX to real stacks. At some cost, Shade could perform this function.[8] Instead, application stacks are just made large.

Application `brk` and `sbrk` system calls (which are called by `malloc`, etc.) are simulated by allocating space following the application's data and BSS segments.

All application memory addresses are offset by the same amount. Thus, once the addresses requiring translation have been identified, address translation itself is simple. Some places where address translation is required include:

- initialization of the application stack pointer and stack frame
- "fetching" application instructions during translation compilation
- simulation of application loads, stores, etc.
- simulation of register window overflow, underflow, and flush traps
- simulation of application system calls: arguments and return values may be addresses, and may point to data structures containing addresses

---

8. Catching faults then extending the stack would be unwieldy, given the way signals are now handled, as there would be three potential contenders for rights to `SIGSEGV`. Automatic stack extension otherwise would be acceptable.

3.6.2. Registers

For each translation, host scratch registers are picked to represent target registers. Host register numbers then replace target register numbers in the translated application code. Host registers are lazily loaded from the virtual state structure, then later lazily stored back, but no later than the translation epilogue.

The host registers cache virtual register values from one translated application instruction to the next in order to reduce memory traffic to and from the virtual state structure. When there aren't enough host registers to represent all of the virtual registers used, host registers are reassigned pseudo-randomly.

Currently 15 integer and all 32 floating point host registers are used to hold virtual register values. On average,[9] 29% of application integer register reads require a prior load from the virtual state structure, and 58% of writes require a store. For floating point benchmarks, 4% of floating point register reads require a prior load, and 30% of writes require a store.

These numbers do not reflect the additional loads and stores that accompany calls to user trace functions which expect to find the application register values in the virtual state structure.

Shade itself doesn't perform many floating point operations while simulating or translating, so the virtual floating point registers could (as they once were) be assigned on a long term basis (such as the duration of `shade_run`, spanning many translations), instead of a short term basis (a single translation). But user trace functions may wish to perform floating point operations, and may wish to find the virtual floating point register values in the virtual state structure. The obvious compromise, using long term mapping only when no user trace functions have been specified, is still under consideration.

Also under consideration is a similar scheme in which high use integer registers (e.g. the stack and frame pointers) are mapped longer term.

Special-purpose registers such as the `y` multiply/divide register, and the `fsr` Floating-point State Register are multiplexed. Swapping tends to be relatively infrequent because Shade does not use these registers as often.

3.6.3. Integer Condition Codes

The integer condition codes are multiplexed. As application code is compiled, Shade keeps track of whether or not the host condition codes currently represent (contain) the virtual condition codes. Functions described below are used to transfer host condition code values to or from the virtual state structure as necessary. Since these transfers are expensive, attempts are made to limit them.

The condition codes are restored from the virtual state structure only if the application code uses the condition codes before setting them. Here "use" of condition codes includes calls to user trace functions, which might wish to read the condition codes from the virtual state structure. On average, the condition codes only need to be restored for 18% of the application instructions which use them.

Condition codes are not always saved by the translation epilogue. Shade can sometimes determine that the next application code to be executed will always set the condition codes before using them. On average, the condition codes only need to be saved for 7% of the application instructions which set them.

Shade's condition code optimization is unsafe in one special case related to signal handling. When a signal is delivered, the current condition code values are copied from the virtual state structure into a signal context structure. This structure may be examined or modified by the signal handler. The condition codes are updated from this structure when the signal handler returns. Shade sometimes discards condition codes between translations, so values copied into the signal context structure may be unreliable. Signal handlers usually ignore the saved condition codes, because most asynchronous signals are delivered when the condition codes are already unpredictable. However, a signal handler could use the saved condition codes (or, equivalently, set new condition code values in terms of the old values), and in this special case the condition code optimization is unsafe. Shade's `-ssignal` flag will ensure that signal handlers receive correct condition code values, but it will also reduce performance. Note that condition code restoration is reliable if the signal handler leaves by calling `longjmp` or `siglongjmp`.

Unfortunately, in Version 8 SPARC, user instructions cannot directly read and write the Processor State Register that contains the integer condition codes. Figure 4 shows a function which reads the condition codes by exe-

---

9. "On average" here and below means geometric mean of dynamically weighted values over the SPEC89 benchmarks.

cuting conditional branches. Each conditional branch tests one bit in the host condition codes. If the bit is set, the branch is taken, and an instruction in the delay slot records that the bit was set. If the bit is clear, the branch is not taken, and the instruction in the delay slot is not executed.

```
shade_geticc:
    clr     %o0
    bneg,a  2f              ! if negative, assume Z==0
    or      %o0, 0x40, %o0  ! N - negative
    be,a    2f
    or      %o0, 0x20, %o0  ! Z - zero
2:  bvs,a   3f
    or      %o0, 0x10, %o0  ! V - overflow
3:  bcs,a   4f
    or      %o0, 0x08, %o0  ! C - carry
4:  retl    %o7 + 8, %g0
    stb     %o0, [%vs + vs_icc]
```

Figure 4. Saving condition codes

Since there are 4 condition code bits, there are a total of 16 possible condition code combinations. The condition codes are written by executing one of 16 code fragments[10] which are specially designed to recreate the condition codes (see Figure 5). The value of the condition codes is used as an index into a table of code fragments where each fragment is 2 instructions (8 bytes). The internal representation for the condition codes (established by `shade_geticc`) was chosen to save a 3-bit shift in `shade_seticc`. Four of the condition code combinations, those with both the zero and negative bits set, cannot be produced by user instruction sequences. If they are encountered by `shade_seticc`, Shade will terminate with a diagnostic.

3.6.4. I/O (File Descriptors)

File descriptors (small numbers returned by the kernel to represent open files or I/O devices) are partitioned between the analyzer and application. Several file descriptor values are special: 0, 1, and 2 represent standard input, output, and error output. Shade renumbers the application's file descriptors so that the application sees the special values even though Shade uses different values.

Shade maintains a table that associates virtual (application) file descriptors with real file descriptors. Shade intercepts each application system call that uses or returns file descriptors, and performs renumbering and table updates as needed. The file descriptor mapping mechanism is available to the analyzer so that it can conveniently control the application's I/O.

Since Shade does not currently retain control across a successful application `exec` system call, there will be no Shade around after the `exec` to renumber file descriptors. So prior to the `exec`, Shade moves the virtual file descriptors to their real locations. Should the `exec` fail, the file descriptors are moved back. Shade should arrange for analyzer files to be closed on application `exec`s, and application files to be closed on analyzer `exec`s, but doesn't. The analyzer can arrange this if necessary.

3.6.5. Signals

Signals can arise for a variety of reasons, and can be delivered either to the application or the analyzer. Signals may also arrive at any time, during execution of Shade, the translations, or the analyzer. Note that in Shade, signals are handled according to ownership rather than time of delivery or cause.

Shade does not reserve any signals, so an analyzer can handle any signal it wishes. Analyzers typically use signals for checkpointing and sampling. All signal numbers are special, and analyzers and applications may use some of the same signals (e.g. hangup, interrupt, alarm clock). Shade prevents applications from interfering with

10. The instruction sequences were discovered by brute force search of condition code setting operations and a small set of promising operand values.

```
shade_seticc:
    ldb     [%vs + vs_icc], %o0
    set     1f, %o1
    jmpl    %o1 + %o0, %g0
    sub     %g0, 1, %o1
1:
    retl; addcc  %g0, 1, %g0    ! ----
    retl; subcc  %g0, %o1, %g0  ! ---C
    retl; taddcc %g0, 1, %g0    ! --V-
    retl; tsubcc %g0, %o1, %g0  ! --VC
    retl; addcc  %g0, 0, %g0    ! -Z--
    retl; addcc  %o1, 1, %g0    ! -Z-C
    retl; tsubcc %o1, %o1, %g0  ! -ZV-
    retl; taddcc %o1, 1, %g0    ! -ZVC
    retl; addcc  %g0, %o1, %g0  ! N---
    retl; addcc  %o1, %o1, %g0  ! N--C
    retl; taddcc %g0, %o1, %g0  ! N-V-
    retl; taddcc %o1, %o1, %g0  ! N-VC
    ba badicc; nop              ! NZ--
    ba badicc; nop              ! NZ-C
    ba badicc; nop              ! NZV-
    ba badicc; nop              ! NZVC
```

Figure 5. Restoring condition codes

the analyzer's signal handling. Where possible, Shade arranges to do so in a way that the application doesn't notice.

Shade intercepts `sigvec` system calls made by the analyzer and records the given signal number. The analyzer then retains exclusive rights to that signal for the remainder of the analyzer run. All signal-related system calls made by the application are intercepted. Shade silently subverts any application requests that would impact the signal handling state (signal handler and flags, blocked signal mask, signal stack) of analyzer-owned signals.

Analyzer signals and their handling are usually invisible to the application, but they can interrupt an interruptible application system call, even when the application has arranged for that signal to be blocked or ignored. Unfortunately, restarting the application system call would be difficult or impossible in most cases.

Signals destined for the application are delivered to a Shade signal handler that notes the signal, and returns control to the interrupted instruction. Eventually, control returns to the main loop, which checks for pending signals. When the main loop detects a signal, it invokes the signal handler instead of the next instruction.

When a signal is delivered, chaining is turned off entirely to reduce the signal handling latency for future signals. The TC and TLB are flushed. Translation chaining is disabled so that a test for pending signals can be made between each translation. A Shade run-time option (`-asignal`) allows the user to disable chaining from the beginning to prevent a delay in handling the first signal to arrive.

Signals arising directly from the execution of an application instruction (e.g. loading or storing unmapped or unaligned data, or division by zero) can often be handled correctly if they occur in Shadow translations. In these cases, Shade will abort execution of the translation, redirect control to the main simulator loop, and signal handling will proceed as described above. Otherwise, a diagnostic is printed for these signals and the faulting instruction is re-executed, which causes the program to terminate since the signal that would be raised is blocked. A Shade run-time option (`-ssignal`) causes Shade to always compile Shadow translations.

Shade ignores the execute permissions associated with application code in memory. If the application code is unreadable, Shade will fault as it reads the code, as it attempts to compile a translation.

3.6.6. Register Windows

SPARC's windowed register file cannot be partitioned (some windows for the application, some for the analyzer) at the user level. The register file could have been multiplexed at some cost involving system calls and/or window overflow and underflow traps. The simplest solution seems to be to consign the host register file to Shade and the analyzer, and simulate the register file as it is used by the application. It is particularly desirable to avoid application window overflow and underflow traps if/when Shade is run on a bare machine with a kernel as the application.

Shade simulates a register file with one window. Therefore, every `save` causes an overflow and every `restore` causes an underflow. Register values are transferred between the virtual state structure and the appropriate save area on the application's stack.

An interesting alternative might be to simulate a virtually infinite register file. In this case the virtual state structure would be huge and no `save` instruction would cause an overflow, nor `restore` instruction an underflow. However, we estimate that, on average, this scheme would save only about 0.2 host cycles per simulated application instruction.

User code is rarely conscious of the number of register windows, but system code pays particular attention to it. It is expected that if kernel applications are someday run under Shade, that the analyzer will be able to specify the number of register windows.

3.6.7. Fork, Exec, and Wait

Shade handles an application `fork` system call by executing a `fork` system call. This produces two separate processes, parent and child, each containing a copy of Shade, the analyzer, and the application.

In the parent process, a parent application process is simulated, traced, and analyzed; in the child process, a child application process is simulated, traced, and analyzed. Unless other arrangements are made, the separate copies of the analyzer cannot share information, and may conflict with each other. A typical problem is that each copy of the analyzer writes results to the same file.

Application `vfork` system calls are converted to `fork` system calls, which may cause some applications to fail under Shade.

Shade handles an application `exec` system call by executing that system call. Shade, the analyzer and the application are all overwritten by the new application program. Shade might have arranged for a new copy of Shade and the analyzer to be started to simulate, trace, and analyze the new application, but doesn't. Or, Shade might have arranged for the new application to overlay the old application (as if `shade_load` had been called), but doesn't. So the new application must be able to run native on the host system.

Shade handles an application `wait` system call by executing that system call. Both the application and analyzer may have outstanding child processes. Unless the process is waited for by specific process ID, these processes are indistinguishable and the analyzer may complete a `wait` for the application's process, or vice versa.

One possible solution to this would be to intercept analyzer and application `fork` system calls and (in the parent) record who made the call, along with the returned child process ID. Application and analyzer `wait` system calls would also be intercepted. If, say, a `wait` made by the analyzer yielded information about an application's process, that information would be recorded against a future `wait` system call by the application, and the analyzer's `wait` would be retried. Integrating this scheme with `SIGCH[I]LD` signal handling might be messy.

3.6.8. Resource Limits

Shade simulates application `getrlimit` and `setrlimit` system calls to prevent the application from modifying the analyzer's resource limits (maximum CPU time, etc.). However, shade does not attempt to enforce the application's resource limits; Shade just maintains a copy of the analyzer's resource limit values for the application to manipulate.

Before Shade executes an application `exec` system call, the application's resource limits are installed as the real resource limits, except that Shade refuses to reduce the maximum (hard limit) value for any resource limit. If the `exec` fails, the previous resource limits are reinstated.

3.6.9. Other Conflicts

The following per-process state or resources are shared between Shade, the analyzer, and the application without special handling.

- process ID and parent process ID
- real and effective user and group IDs, and supplementary group IDs; for simplicity and security, Shade refuses to load setuid or setgid applications
- process group, TTY process group, session ID, and control terminal
- file creation mode mask
- current working directory and root directory
- scheduling priority
- interval timers
- resource usage (CPU time used, etc.)
- anything to do with System V IPC messages, semaphores, or shared memory
- `ptrace` flag

4. Performance

In this section we give measurements for various configurations running various programs. The *standard organization* used in many of the following tests is:

- The TC holds 2^{20} instructions (4MB).
- The TLB contains 2^{13} (8K) lines, each containing 4 address pairs (256KB).

The benchmarks used for this section are the SPEC89 *001.gcc1.35* and *015.doduc* benchmarks. They were compiled using SPARC Compilers 1.0 with the options:

```
-cg89 -O4 -libmil -dalign
```

The tests were run on a SPARCstation 2.

Six Shade analyzers, each performing a different amount of tracing, were used:

`icount0`
: no tracing, just application simulation (The application runs to completion in one call to `shade_run`.)

`icount1`
: no tracing, just update the traced instruction counter (`%ntr`) to permit instruction counting

`icount2`
: trace PC for all instructions (including annulled); trace effective memory address for non-annulled loads and stores (This corresponds to the tracing required for cache simulation.)

`icount3`
: same as `icount2` plus instruction text, decoded opcode value, and, where appropriate, annulled instruction flag and taken branch flag

`icount4`
: same as `icount3` plus values of all integer and floating point registers used in instruction

`icount5`
: same as `icount4` plus call an empty user trace function before and after each application instruction

The analysis performed by each of these analyzers consisted solely of counting the number of instructions executed. In a realistic analyzer, analysis would dominate the run time.

Figure 6 shows how much slower applications run under Shade than they run native (directly on given hardware). The *inst* column shows the number of instructions that were executed per application instruction. Instruction counts were gathered by running the Shade `icount1` analyzer on Shade running the indicated analyzer on the indicated benchmark. The *time* column shows the CPU (user + system) time ratio.

app	icount0		icount1		icount2		icount3		icount4		icount5	
	inst	time	inst	time	inst	time	inst	time	inst	time	inst	time
gcc	5.51	6.13	5.85	6.60	8.84	14.32	13.50	21.71	15.51	31.21	63.74	84.17
doduc	2.75	2.84	2.94	3.14	5.52	8.78	9.44	14.03	11.45	24.06	36.25	60.30

Figure 6. Dynamic expansion: instructions and CPU time

Figure 7 shows how much larger dynamically (i.e. weighted by number of times executed) a translation is than the application code it represents. The sizes are in instructions. Since portions of most translations are conditionally executed, the translation code space expansion values shown in this table are typically larger than the executed code expansion values shown in the previous table. For `icount0` and `icount5` on *gcc*, a larger share of the instructions are executed outside of the TC in the translation compiler, simulation functions, and the analyzer.

app	input size	output size					
		icount0	icount1	icount2	icount3	icount4	icount5
gcc	5.1	20.1 4.7x	25.5 6.2x	40.6 9.1x	66.8 14.8x	76.9 16.7x	192.8 39.9x
doduc	12.5	33.2 4.1x	38.7 5.1x	73.2 8.0x	126.0 13.2x	152.5 15.1x	427.0 38.6x

Figure 7. Code translation expansion

Figure 8 shows the percentage of total run time spent in various phases of execution. *Compiler* is the time spent in the translation compiler. *TC* is the time spent executing code in the Translation Cache itself. *Sim* is the time spent in functions which are called from the TC to simulate, or assist in simulating application instructions. *Analyzer* is time spent in the user's analyzer, including user trace functions which are called from the TC. This information was obtained with standard profiling (`cc -p; prof`). The standard TC and TLB size and organization were used.

The time distribution is a factor of several things. Better optimization takes longer and produces faster running code, both of which increase the percentage of time spent in code generation. A small TC increases the frequency with which useful translations are discarded. A small or ineffective TLB increases the frequency with which useful translations are lost. Translations that collect a lot of information take longer to run, and thus reduce the percentage of time spent in simulation functions, even though their absolute running time is unchanged. All of the analyzers are trivial, though `icount5` supplies null functions that are called before and after each application instruction.

app	location	icount0	icount1	icount2	icount3	icount4	icount5
gcc	Compiler	8.77%	10.14%	5.82%	4.59%	3.86%	25.05%
	TC	51.13%	52.56%	74.62%	82.22%	86.33%	61.26%
	Sim	39.00%	36.09%	19.01%	12.83%	9.55%	4.97%
	Analyzer	0.00%	0.03%	0.01%	0.00%	0.00%	8.61%
doduc	Compiler	0.22%	0.35%	0.16%	0.11%	0.08%	0.06%
	TC	80.69%	81.56%	91.50%	95.20%	96.37%	87.87%
	Sim	19.04%	17.97%	8.32%	4.67%	3.55%	2.11%
	Analyzer	0.00%	0.07%	0.00%	0.00%	0.00%	9.96%

Figure 8. Run-time execution profile summary

Figure 9 shows the frequency of TC flushes. The frequency is a function of the TC size and indirectly a function of the rate at which translations are lost from the TLB. The data in Figure 9 was collected with varying TC sizes and the standard TLB size and configuration. The standard configuration is shown in **bold**.

Figure 10 shows the TLB behavior under various configurations.

- Each program is run with the translation chaining optimization turned on and off. The *Ch* column contains "Y" if translation chaining is turned on.

app	TC size	icount0	icount1	icount2	icount3	icount4	icount5
	256KB	1338	1952	4120	10862	13801	110389
	512KB	366	656	1344	2878	3582	20767
gcc	1MB	4	50	377	1060	1244	5306
	2MB	0	0	5	197	317	1628
	4MB	**0**	**0**	**0**	**0**	**2**	**539**
	256KB	5	5	27370	159152	164258	261997
	512KB	0	0	1	6123	10820	27332
doduc	1MB	0	0	0	1	1	10949
	2MB	0	0	0	0	0	17
	4MB	**0**	**0**	**0**	**0**	**0**	**0**

Figure 9. TC/TLB flushes per 10^9 application instructions

- Each program in Figure 10 was run with several TLB sizes and organizations. The standard configuration is shown in **bold**. Each TLB entry consists of a 4 byte virtual machine address and a corresponding 4 byte host machine address. So the amount of memory consumed is 8 times the number of entries shown in the *TLB Entries* column.
- The TLB associativity was also varied, with the TC size held constant.
- The average search length (*Len* column) shows how many entries on a line are examined to discover a hit or a miss. For a direct-mapped TLB, the search length is trivially one. For associative organizations, the average search length is dominated by the optimization that moves recent hits to the front of each TLB line. When a translation is moved to the front of the TLB line, it is usually used several times before it is displaced. Since the miss rate is low, the average search length for hits is almost identical to the overall search length.

  For a fixed-size TLB, increasing the associativity reduces the conflict rate. For example, a fully-associative TLB with this protocol would only replace entries when the TLB line was full, and the least recently used entries would be replaced. However, increasing the associativity increases the number of translations that map to a given line. That increases the probability that two "active" translations will map to the same line and thus contend for the first slot in the TLB.

  Note that the main loop can efficiently check the first TLB slot, but if this fails, a relatively expensive call to `shade_trans` must be made to complete the TLB lookup.
- The miss rate (*Miss %* column) is the percentage of TLB lookups that failed. On each miss, a new translation is compiled. There are always at least as many misses as there are basic blocks (translations) in the program. The number of misses increases when cached translations are deleted from the TLB to make space for new translations.
- For each TLB miss, a new translation is compiled and entered in the TLB. The replacement rate (*Lost %* column) is the percentage of new translations that cause other translations to be discarded. The replacement rate is affected by the TLB associativity, and the TC flush rate (since the TLB is flushed along with the TC). For these experiments, the TC was never flushed. If the replacement rate were 0% then all misses could be attributed to executing code that hadn't been executed before. If the replacement rate were 100% then all misses could be attributed to TLB conflicts. A high replacement rate means that translations are being recompiled frequently.

  Typically, more than 90% of the replaced translations are needed again later, causing recompilation. Greater associativity or a secondary "TLB victim cache" [Jouppi90] could reduce replacement and thus recompilation.
- The translation chaining optimization reduces the number of TLB references. The miss rate increases because translation lookups always miss when the translation is first referenced (cold misses) and because the number of TLB references goes down. However, the absolute number of misses and replacements stays nearly the same. Chaining reduces the demand for TLB entries, which tends to reduce the number of lost TLB entries. The average search length improves, however, because only unchainable, hence fewer, translations are competing for the head of the list.

Figure 11 shows the average number of instructions that are executed by the code generator in order to generate one machine instruction. The number of instructions per instruction in the code generator is a function of the instruction set architecture of the host and target machines and the level of optimization. The statistics were collect-

				4-Way Assoc			2-Way Assoc			Direct Map	
app	Ch ?	TLB Entries	TLB Refs	Len	Miss %	Lost %	Len	Miss %	Lost %	Miss %	Lost %
gcc	N	16K	244M	1.097	0.091	3.130	1.037	0.105	4.061	0.445	4.503
	N	32K	244M	1.063	0.102	0.650	1.033	0.097	2.926	0.451	4.408
	N	64K	244M	1.048	0.104	0.048	1.028	0.103	0.981	0.469	4.338
gcc	Y	16K	31M	1.032	0.716	0.842	1.008	0.654	1.479	0.562	3.000
	Y	32K	31M	**1.015**	**0.806**	**0.116**	1.005	0.761	0.475	0.685	1.651
	Y	64K	31M	1.007	0.819	0.012	1.002	0.808	0.099	0.758	0.868
doduc	N	16K	103M	1.023	0.003	0.001	1.014	0.004	1.297	0.039	2.786
	N	32K	103M	1.015	0.003	0.000	1.012	0.003	0.009	0.038	2.744
	N	64K	103M	1.012	0.003	0.000	1.012	0.003	0.003	0.038	2.744
doduc	Y	16K	7.8M	1.000	0.037	0.001	1.000	0.037	0.058	0.035	0.520
	Y	32K	7.8M	**1.000**	**0.037**	**0.000**	1.000	0.037	0.009	0.036	0.378
	Y	64K	7.8M	1.000	0.037	0.000	1.000	0.037	0.003	0.037	0.370

Figure 10. TLB behavior

ed by running Shade on Shade, profiling just the code generator to determine the number of instructions executed in the code generator, and by having the profiled Shade report the number of instructions that it generated.

app	icount0	icount1	icount2	icount3	icount4	icount5
gcc	179.7	171.25	127.2	94.0	87.5	51.4
doduc	245.4	271.2	162.6	111.9	102.1	58.9

Figure 11. Code generator instructions per instruction generated

5. Cross Shades

In the previous section we focused on the Shade (subsequently referred to as Shade-V8.V8) for which the host and target architectures were both Version 8 SPARC, and for which the host and target operating systems were both SunOS 4.x [SunOS4]. Four other Shades have been developed. The first (Shade-MIPS.V8) runs UMIPS-V [UMIPSV], MIPS I [Kane87] binaries, and the second (Shade-V9.V8) runs SunOS 4.x, Version 9 SPARC [SPARC9] binaries. The host system for both is SunOS 4.x, Version 8 SPARC. There are also versions of Shade-V8.V8 and Shade-V9.V8 where both the host and target operating systems are Solaris 2.x [SunOS5].

All of these Shades are at least complete to the extent that they can run SPEC89 binaries compiled for the respective target systems.

5.1. Shade-MIPS.V8

Shade-MIPS.V8 was developed to provide the custom tracing capabilities of Shade for MIPS binaries. Given Shade-V8.V8 and ready access to SPARC systems, SPARC was the natural choice for host architecture.

Little attention was paid to simulation efficiency, beyond the efficient simulation techniques already used in Shade. Also, Shade-MIPS.V8 only simulates instructions and system calls used in SPEC89 binaries. On average, Shade-MIPS.V8 executes about 10 SPARC instructions to simulate a MIPS instruction. We wouldn't expect more than a factor of two increase in simulation efficiency to be easily achieved with Shade.

As a rule, MIPS instructions are straightforward to simulate with just a few SPARC instructions. This is possible because both the MIPS and SPARC architectures are RISC architectures, both support IEEE arithmetic, and the MIPS architecture lacks integer condition codes. The rest of this section describes some of the remaining difficulties, solutions, and compromises.

Though MIPS systems support both big-endian and little-endian byte ordering [James90], SPARC only supports the former. The simulator currently runs only code that has been compiled for MIPS systems running in big-endian mode. To support little-endian byte ordering, Shade-MIPS.V8 would have to reformat the data for multi-

byte loads and stores, which would significantly increase the cost of these instructions.

A similar swapping problem exists for floating point registers. The most significant half of double precision register number N is numbered N+1 on MIPS and N on SPARC. Shade-MIPS.V8 must translate floating point register numbers accordingly.

The immediate field size for common operations is 16 bits for MIPS, but only 13 bits for SPARC. Where target immediates do not fit in 13 bits, one or (most likely) two SPARC instructions are used to place the immediate value in a host scratch register.

The MIPS integer divide instructions perform a divide which provides a remainder. SPARC integer divide instructions aren't implemented in hardware on older systems, and don't provide a remainder. Shade-MIPS.V8 simulates integer divide by calling the standard divide and remainder library functions.

In place of integer condition codes, the MIPS architecture employs values stored in general purpose integer registers. This frees Shade-MIPS.V8 from multiplexing condition codes as is done when simulating SPARC instructions. Some MIPS instructions assign values to registers based on comparison results or branch based on a register value. Shade simulates them in the obvious way, using SPARC compare, conditional branch, and set instructions.

For signed integer add and subtract instructions, Shade-MIPS.V8 does not check for overflows that would cause exceptions on MIPS systems.

The MIPS unaligned memory operations (load/store word left/right) have no equivalents in SPARC instructions and are simulated by calls to special-purpose functions.

The MIPS floating point control register is very similar to the SPARC floating point status register, a fortunate result of IEEE arithmetic support. The bits are arranged differently though, and translations must be performed.

The MIPS floating point instructions which convert floating point to integer values respect the current rounding mode, but the corresponding SPARC instructions always round towards zero. So these MIPS instructions are simulated by calling a math library function.

The object file formats differ between SVR4 and SunOS 4.x, so a different, more complicated, program file reader is required.

Shade-MIPS.V8 intercepts all the application's system calls. Many of the SVR4 system calls have direct equivalents under SunOS. Some have evolved, leaving behind compatibility functions (e.g. `signal` and `time`) which are suitable for simulation purposes. Some system calls require data translation of varying complexity, for instance `stat` (easy) or `ioctl` (hard and virtually unimplemented). Fortunately, less portable system calls are less likely to appear in benchmarks.

5.2. Shade-V9.V8

Shade-V9.V8 simulates a V9 SPARC target and runs on a V8 SPARC host. The principal problems of simulating V9 applications on V8 hosts are wider integer registers and additional condition codes. Simulating a 64-bit address space would be a problem, but so far it has been avoided. The new V9 instructions themselves do not present much in the way of additional problems, although there are a lot of additional instructions. Shade-V9.V8, like Shade-V8.V8, currently simulates only user-mode instructions.

As a rough measure of relative *simulation* complexity, consider that, given Shade-V8.V8, it took about 3 weeks to develop Shade-MIPS.V8 and about 3 months to develop Shade-V9.V8 to the point where each could run SPEC89.

Generally Shade generates a short sequence of V8 instructions for each V9 instruction. Notable exceptions to this are the 64-bit multiply and divide operations, which Shade compiles as calls to simulation functions. Figure 12 shows the translation body fragment for a V9 `add` instruction.

V9's 64-bit registers are simulated with V8 register pairs. This doubles the virtual state structure memory traffic. Additional traffic results from being able to simultaneously represent only half as many target machine application registers with the same number of host scratch registers.[11]

---

11. Shade-V9.V8 may thus be starved for registers. On register-intensive translations, using `save` and `restore` within translations could alleviate some register pressure problems.

```
ldd     [%vs + vs_r1], l0
ldd     [%vs + vs_r2], l2
addcc   l1, l3, l5        ! add lsw
addx    l0, l2, l4        ! add msw
std     l4, [%vs + vs_r3]
inc     4, %vpc
```

Figure 12. Shade-V9.V8 translation body

V9 SPARC has two sets of condition codes. Each instruction that sets integer condition codes sets one set based on the low order 32 bits of the result (just as in V8), and another based on the full 64 bits of the result. The host integer condition codes are often required (as in the `add` example above) to simulate 64-bit operations which themselves do not involve condition codes. All this keeps the `shade_[gs]etcc` functions very busy.

For these reasons, wide registers and more condition codes, Shade-V9.V8 is less efficient than Shade-V8.V8, and less efficient for integer than floating point applications. On average, Shade-V9.V8 simulates V8 (*sic*) integer SPEC89 benchmarks 12.2 times slower than they run native, and V8 floating point SPEC89 benchmarks 3.3 times slower. Shade-V8.V8 simulates these same benchmarks 6.2 and 2.3 times slower than they run native, respectively.

V9 supports more floating point registers and floating point condition codes than V8. As compilers take advantage of these, Shade-V9.V8's floating point simulation performance will decline.

In V9, memory addresses for floating point doubleword and quadword loads and stores need only be word aligned, whereas in V8 they must be doubleword and quadword aligned. So these operations are simulated by breaking them down into word operations.

In V9, the floating point condition codes may be used by a conditional branch instruction immediately following the floating point compare instruction which set them. In V8, the floating point compare and branch instructions must be separated by at least one other instruction. Shade-V9.V8 sometimes generates a `nop` instruction following the floating point compare instruction; usually Shade generates other simulation or tracing code that makes this unnecessary.

Programs may manipulate 64-bit pointers under Shade-V9.V8, but when it comes time to actually access memory (load, store, register indirect jump, system call), Shade-V9.V8 ignores the upper 32-bits of the memory address. It can do this since by construction the application only has access to the the lower 4GB of virtual memory. To catch application addressing bugs, Shade-V9.V8 could test the upper 32 address bits to make sure they are zero, but doesn't. Shade-V9.V8 will run slower if and when it needs to simulate a full 64-bit address space.

The host kernel has a 32-bit interface. Only the low order 32 bits of parameters are passed to system calls, and the upper 32 bits of return values are cleared.[12] Note that the SunOS 4.x kernels indicate success/failure via the integer condition codes which user code interfaces interpret to return a 64-bit −1 on failures. Structures (e.g. `sigcontext`) passed to/from the kernel through memory are more problematic.

Shade provides two options to support simulation of both 32-bit and 64-bit binaries. The first option selects the size (32 or 64 bits) of integers and pointers as they appear in the initial stack. This affects the application runtime startoff code, which may let the effects spread to `main` and `environ`. The second option specifies the width (32 or 64 bits) of the stack register save area for `save` and `restore` instructions. This affects the stack frame layout.

12. In rare cases (e.g. the `pathconf` system call), sign extending the return value from 32 to 64 bits may be preferable to zero extending it.

6. Shade-X.Y

The previous sections describe the specific implementation of Shade for a few target systems and one host system. This section considers what is needed to build a general "Shade-X.Y," namely a Shade-like simulation and tracing tool that simulates target system X on host system Y. Unless explicitly stated otherwise, the implementation techniques presented in this section have not been used in the current Shade implementations.

6.1. The Architecture of Shade

Shade is a virtual machine implemented using a lazy incremental compiler. The retargeting issues are the same as for a regular compiler, with the additional requirement that code needs to be produced quickly in order for the system to be effective.

Shade parses the target system's machine language. Target machine basic blocks are the usual "units" of compilation, although Shade will sometimes compile longer or shorter fragments. Shade makes a pass to copy application code and count traced instructions. It then analyzes each instruction in the context of the basic block information, determining what analysis data should be collected about each instruction, and makes calls into the code generator. The code generator produces native machine code for the host system.

In Shade, both the front end and back end are ad hoc. Although Shade has been ported to several target machines, it is not a retargetable compiler because the interface between the front end and the back end depends on both the target and host languages.

Consider a target machine that supports 32-bit immediate constants, such as the VAX [DEC81] or 68000 [Motorola84], and consider a host machine that supports 16-bit immediates. When the front end encounters an immediate, the back end must check the immediate to decide whether it can generate a 16-bit immediate, or whether it must synthesize a 32-bit immediate with several instructions. If the target machine is changed to one that has immediates of 16 bits or less, such as MIPS, the back end will never need to synthesize 32-bit immediates. If the target machine is changed to one with immediates smaller than 16 bits, then the back end may sometimes be able to collapse a `sethi` and some instruction with an immediate into just one instruction.

In practice, separating the front end and back end causes inefficiency because we are not able to statically specialize the dynamic compiler. In the above example, for instance, a general-purpose back end will always test immediates to see if they are too large for the host machine's immediates. For target machines with immediates that are smaller than the host's, the test will always fail, and is thus redundant. Likewise, a general-purpose back end will test to see if pairs of instructions can be collapsed. For targets with larger immediates, the test for collapsing pairs of instructions always fails.

In principle, the front end and back end can communicate in a way that is independent of the host or target machines (but probably not both; the *UNiversal COmputer Language* or *UNCOL* problem [Conway58] is generally regarded as unsolvable). For example, the front end can be statically parameterized by the immediate size of the host, and can be compiled to call different back end routines depending on the size of the target machine immediates. We have not yet tried to refine the interface to improve portability.

6.2. Optimizations

Shade replaces a traditional simulator's inner loop with translations that are customized to the simulation. The resulting code is larger, but is specialized to the application, so Shade executes fewer instructions than a traditional simulator.

The cost of Shade's dynamic compiler is a part of the cost of running Shade. For a static compiler, the time to perform the optimization is irrelevant to the performance of the final code. With a dynamic compiler, payback is achieved only when cost of the optimization is less than the speedup of the resulting code.

Some dynamic compiler optimizations reduce the number of instructions which are executed to run the program, but make translations so much larger that the optimizations are "pessimizations" because of the increased code size. There are various reasons for the slowdown: more optimization takes more time; generating larger code takes more time; larger translations fill the TC faster, reducing the effectiveness of both the TC and TLB and increasing the rate of recompilation; using a larger TLB or TC increases virtual memory demands; and large code hits less frequently in the host machine's instruction cache.

Which optimizations are most valuable? With tracing off, reads and writes of the virtual state are a large part of the overhead. With tracing on, tracing dominates execution time.

Shade generally takes an instruction-by-instruction view of the target code. Better optimization could speed up the untraced code, but overly aggressive optimizations interfere with the simulation by, e.g., discarding information that is not needed for running the program but which is needed for collecting tracing information.

6.3. Condition Codes

This section describes machine features that make condition code simulation thorny and describes implementation strategies for reducing the costs.

6.3.1. Machine Variations

If both the host machine and the target machine have condition codes, the host machine's condition codes need to be saved and restored frequently. In simple cases, condition codes are needed by Shade only at translation boundaries to check for buffer overflow, so there is at most one save per translation, and, thus, at most one restore per translation. In harder cases, such as Shade-V9.V8, condition codes are used in emulating target instructions, so condition codes may need to be saved and restored several times per translation.

Target machines with no condition codes have a compare-and-branch instruction and are easy to simulate on a host machine that has condition codes. The simulation may be less efficient as it sometimes replaces a single compare-and-branch with two instructions, a compare and then a branch. On some machines, e.g., MIPS, compare-and-branch instructions only perform negative/positive and zero/nonzero tests. Therefore, the target's compare-and-branch is frequently preceded by a subtraction and the host's simulation of that subtraction will also set the host condition codes appropriately.

Host machines without condition codes must simulate the condition codes each time they are needed. Since the condition codes are set by, e.g., a subtraction, and since checks for overflow and carry are less common, it is often sufficient to use compare-and-branch on the saved result of the arithmetic operation.

The host machine may implement condition codes but set them by different rules than the target machine. Rather than simulate them fully, it may be best to save the values and operation that set the condition codes. Later, when it is known what condition codes are needed, just those condition codes can be simulated.

It may be hard to save and restore condition codes efficiently. In Version 8 SPARC, for example, the condition code register cannot be read nor written directly, so multi-instruction sequences are used instead. VAX condition codes are saved and restored with special instructions, but the special instructions are slower than other operations.

Most SPARC instructions which set the integer condition codes also come in a form which do not set condition codes. Thus, target code generally sets condition codes only when they are needed. On some machines, nearly every instruction sets the condition codes. The translation compiler must analyze the block to determine which places set condition codes gratuitously, and which produce values that must be saved.

On some machines, the condition codes are set uniformly by all instructions. On others, e.g., the VAX, an integer add (which sets all condition codes) followed by bit clear (which leaves the carry bit unchanged) leaves the condition codes set as a function of two instructions.

6.3.2. Inter-Block Analysis of Condition Codes

If Shade can determine that the following block always sets the condition codes before use, the current block does not save condition codes that it sets. Analyzing branch targets this way is a special case of inter-block register analysis. The payoff is particularly high because saving and restoring condition codes is particularly expensive. Shade-V8.V8 uses this optimization.

Inter-block analysis also potentially allows the current translation to simulate only the condition codes it uses, rather than simulating all condition codes for all uses.

6.3.3. Partial Restoration of Condition Codes

When the host restores condition codes for, e.g., a conditional branch, only certain condition code bits are needed. For example `bneg`, branch on negative, only needs to test if the negative (N) condition code is set. Rather than setting all host machine condition codes, just the needed codes can be checked. Figure 13 shows the host's zero (Z) condition code being set if and only if the target's negative condition code was set.

```
ld    [%vs + vs_icc], %tmp     ! load target cc's to a register
andcc %tmp, 0x40, %g0          ! extract the N bit
bnz   ...                      ! branch if N was set
```

Figure 13. Inlined partial condition code restoration

Restoring any single condition code takes one masking instruction, `andcc`. Combinations of set and cleared bits, such as carry (C) set and overflow (V) clear, require two instructions: `and` to extract and `xorcc` to test. Some condition code expressions are harder; the expression for `bg`, branch on greater, is: $\neg(Z+(N \oplus V))$.

In-line expanding code for partial restoration, as above, eliminates call/return overhead. It also enlarges each translation that has condition code restoration.

6.3.4. Deferred Simulation of Condition Codes

If it is particularly hard to save the condition codes, it may be profitable to save data to recreate them on demand. In general, this requires saving a few source values and an operation indicator.

Consider the application fragments in Figure 14a, where `first` sets the condition codes for use by `second`.

```
first:
    subcc   %i0, %i1, %i2
    ba,a    second
     ...
second:
    bneg    third
    nop
```

Figure 14a. Application sets the condition codes

Shade-V8.V8 generates a `first` translation that makes a call to `shade_geticc` to save the condition codes in the virtual machine state. In the translation for `second` there is a call to `shade_seticc` to restore the condition codes for `bneg`.

An alternative `first` translation body (see Figure 14b) saves values and an index that indicates which operation `first` uses to set the condition codes. The translation for `second` computes the condition codes by calling `shade_opcc` (see Figure 14c), which loads the two values and repeats the operation that `first` uses to set the condition codes.

```
    st      %i0, [%vs + vs_val0]      ! save source values
    st      %i1, [%vs + vs_val1]      !  ...
    add     %g0, 0x08, %scratch       ! build index into opcode tbl
    st      %scratch, [%vs + vs_op]   ! .. and save it
    subcc   %i0, %i1, %i2             ! set %i2
```

Figure 14b. Saving values and operation for later use

```
shade_opcc:
    ld      [%vs + vs_val1], %o1      ! load one source value
    ld      [%vs + vs_op], %o2        ! load the operation number
    set     1f, %o0
    jmpl    %o0 + %o2, %g0            ! .. and jump to that entry
    ld      [%vs + vs_val0], %o0      ! load other source value
1:
    retl; addcc %o0, %o1, %g0         ! add: opcode index = 0x00
    retl; subcc %o0, %o1, %g0         ! sub: opcode index = 0x08
    retl; andcc %o0, %o1, %g0         ! and: opcode index = 0x10
     ...
```

Figure 14c. Restoring the condition codes

The `shade_opcc` routine uses the index to dispatch to an instruction that sets the condition codes. `first` saves an index but could save the address of the restore fragment instead.

For Shade-V8.V8, deferred simulation would be marginally better in dynamic instruction counts. However, translations that save and restore condition codes are larger, have more memory traffic, use more registers, and don't save that many instructions: 5 vs. 11 to save, 8 vs. 7 to restore. Partial restoration can further reduce the `shade_geticc`'s 7 instructions to restore condition codes, but cannot be used with deferred simulation. Finally, inter-block use/set analysis minimizes the number of translations that need to save and restore the virtual condition codes, so small differences in save/restore performance are relatively unimportant.

The `shade_geticc` and `shade_seticc` functions can save and restore condition codes using a single byte store and a single byte load. In Shade-V8.V8, deferred simulation would require two word stores and two word loads. For Shade-V9.V8, values are 64 bits, further increasing the memory traffic and register pressure. Furthermore, 64-bit load and store instructions may take longer to complete. Finally, the operations in `shade_opcc` that set the 64-bit condition codes are multi-instruction sequences. Note, however, that one deferred simulation can replace two calls to `shade_geticc` if both 32-bit and 64-bit condition codes need to be saved.

On other machines the costs will be different. The above is given as an example to show the kind of analysis that is needed.

6.3.5. Hoisting Branch Targets

If condition codes are set in one translation and used immediately in the successor, then the branch target can be hoisted into the translation that sets the condition codes. Potential costs of this approach include growth in the translation size and translations with many epilogues.

6.4. Register Assignment

Shade generally assigns host registers to target registers for one translation, which is called *local* allocation. In the best case, each target register is read at most once per translation from the virtual state structure, and is written at most once per translation. Some registers, such as the target program counter, are used so frequently that it almost always helps to allocate host registers just for them. These registers are allocated the same in all translations; this is called *global* allocation.

Most target registers are used frequently in certain regions of a program, but usage patterns change over the program, so global allocation of these registers hurts more than it helps. The translation compiler might instead perform *regional* allocation. When control passes between translations using the same policy, no save/restore traffic is needed for regionally-allocated registers. At policy boundaries, all non-global registers are saved to the virtual state, the same as with local allocation.

With a register-starved target and a register-rich host, it may be possible to do global allocation and put all target registers in host registers, eliminating most reads and writes of the virtual state. With a register-starved host there are many saves and restores within a translation, so neither regional nor global allocation is generally useful.

Some registers, especially the condition codes, are multiplexed between the host and the target and cannot be allocated across multiple translations. However, modified registers that are reset before use by all successor translations need not be saved. The translation compiler can discover such registers with inter-block analysis. Such *dead store elimination* is particularly valuable for, e.g., condition codes on SPARC, where saves are expensive and the condition codes are rarely used outside of the translation in which they are set.

6.5. Byte Order

If the byte order of the host and target machines is different, multibyte memory accesses must simulate byte reordering in order to get values loaded into host machine registers properly. Data cannot be stored in reversed order in the program for two reasons: First, the program may use a multibyte quantity as both a single multibyte object and as an array of bytes. Second, some values are created statically by the compiler and thus are in the "wrong" byte order. It is hard to determine whether a given load/store is for a statically-generated or dynamically-generated number.

Byte order is sometimes called endian-ness [James90], and byte reordering "byte swabbing" [Mitze77]. A simple load instruction in the application code may require 8-10 instructions on the host [James90].

It may be possible to optimize stack-relative loads and stores to use host machine byte-order rather than virtual machine byte-order. Such an optimization will definitely break on arbitrary code, but may work on, e.g., portable C code.

6.6. Alignment

Some architectures (e.g. VAX and RS/6000 [IBM90]) allow multibyte quantities to be read from any address. Others (e.g. MIPS and SPARC) require multibyte loads and stores to be aligned on boundaries which match the size of the data item. Thus, simple loads and stores in the application must sometimes be translated into complicated interpolation sequences.

Some architectures have instructions to assist with unaligned loads.

Simulating unaligned loads is generally expensive enough that it is worth performing tests to avoid simulation. One strategy is to compile two translation bodies: one assumes aligned pointers, the other works with any alignment. The translation checks the pointers on entry and branches to the appropriate body. If the aligned or unaligned case is sufficiently rare, it can be compiled on demand.

Another alternative is to assume that a particular application load or store instruction will use memory addresses with the same alignment from one execution to the next. The compiler examines the alignments of the pointers and compiles the translations to use loads and stores matching those alignments. Each translation has code that checks if the assumed alignments still hold and, if not, compiles a new translation for the new alignment. A new translation can be optimized to the new pointer values [Johnston79], the new translation can be more general and less optimized [SW79], or several versions can be kept and the version that is used depends on the alignments on any given call [May87, HCU91].

Some operating systems allow unaligned loads and stores, at the cost of a trap. If unaligned accesses are rare, the occasional trap may be preferable to constant testing or interpolation.

Note: Alignment and byte order handling code can sometimes be combined.

6.7. Memory Layout

So far, the description of Shade has assumed that address translation is used to map target addresses to host addresses. Another approach, used in Shadow and older Shade implementations, is for the host and target to have identical memory layouts, such that a value at, say, location `0x53` on the target is also at location `0x53` on the host. However, the memory layout may be different across machines in ways that make it impossible to allocate code or data on the host machine in the locations expected by the target machine.

The machines that we are considering generally have three segments: code, data (including BSS and heap), and stack. The address space may also have *holes* that are inaccessible for various reasons. Page size may also affect memory layout.

Ideally, the target machine code and data are allocated at the same location in the host machine as they are located on the target machine. It is generally unimportant where the stack is located. The host machine need not have the same default layout as the target machine in order for the two to be mapped identically. For example, under

SunOS, the data segment begins immediately following (or in the first page immediately following) the code segment. However, using the `mmap` system call, the target machine data for a MIPS application can be mapped to start instead at its normal location, 0x10000000, far from the text.

Sometimes it is impossible to locate the target machine memory at the desired location. For example, under VMS the first 512 bytes of the address space cannot be read, written, or executed by a user process. Thus, memory for a target machine that references low locations cannot be loaded at the proper location. It is then necessary to *transliterate* addresses. For example, code could be shifted to addresses that are all 512 bytes higher and all code segment references can be offset by 512.

Constant data values are often stored in the code segment. If only one segment is offset, then generic loads and stores must check to see what offset is needed. If both code and data are offset, then all references may be offset blindly. With blind offsets, the approach used in Shade, execution is slowed down only slightly because the mapping from target machine addresses to host machine addresses is mostly done by the compiler, and the compiler runs infrequently.

The above discussion of memory layout ignores the simulator. The simulator must also live somewhere in the address space. It is assumed that the host machine will be able to set aside a sufficient part of the address space.

Some applications change page protection to implement garbage collection, distributed shared memory, and so on [LH89]. Memory protection is an OS emulation issue, but deserves special attention here because problems such as differing target and host page sizes can be dealt with as with page accessibility problems. Such problems can usually be handled lazily, running at full speed until the application makes a call to, e.g., change page protection.

In the worst case, memory can be simulated entirely, but at a cost of more than 10 instructions per load and store [Bedichek90a]. A simulation of multiple address spaces requires translation on every application memory reference [Bedichek90a].

A simulator that models multiple address spaces will need to perform address translation on every reference, since a given address will reference different physical storage depending on the address space in which it is issued [Bedichek90a]. Note, however, that Shade performs instruction address translation when translations are compiled, rather then when they are executed.

6.8. Page Size

For many programs, page size is not an issue; either they are unconcerned with page size, or use whatever page size is supplied by the kernel at run time.

One problem area is that binaries are often set up to start the data on the first page following the last page of code. If the host's page size is larger than the target's, the last target code page and first target data page may reside in the same host page. Solutions include offsetting the code and data and changing host protections to allow writes to the last code page.

Another problem area is that some programs will reasonably expect to be able to change page protections on the granularity of the target machine. Whenever the host and target have different mappings for the target pages, the simulation for memory references must include protection checks.

6.9. Floating-Point

There are two issues with floating-point values: One issue is whether the host and target use the same floating-point format. The other issue is whether a given operation will produce the same result on the host and target.

When the host and target use different floating-point formats, it is generally faster to convert target values to host values and perform operations using native host instructions. As with byte ordering, floating-point values are generated statically, so conversions must be performed on every load and store. In most cases, the host can operate directly on the converted values; few applications rely on the internal format of floating-point values. Some libraries rely on the internal representation, so occasionally the target's floating-point format must be emulated, but sometimes only for integer (including bitwise) operations on the floating-point values. When the host and target use different layouts it is often difficult (costly) to simulate the target operations exactly.

When the host and target use the same format, no conversion is needed. Machines that are IEEE-compliant are required to have bitwise identical representations for all numbers and are required to perform rounding the same

way. Furthermore, compliant implementations are required to handle special (infinite or indeterminate) values in a standard way.

IEEE-compliant systems include: IBM RS/6000, MIPS, Motorola 68000 and 88000 [Motorola89], and SPARC. Non-IEEE systems include: IBM 360 family and VAX.

Note that an IEEE-compliant application is guaranteed to be portable. It is not required that all applications running on a given platform are IEEE-compliant.

There may be rounding differences in the conversion between virtual machine and host machine formats. The rounding differences may occur even if the host machine uses a higher-precision representation than the virtual machine.

Note that machines may use more bits of internal precision, so machines with compatible layouts may still produce different answers. Thus, equivalent operations on machines with compatible formats may produce different results.

6.10. Jumps and Linkage

Most machines have a near/fast branch and a far/slow branch. On most machines conditional branches are available only in the near/fast form. Application code generally exhibits good locality, but translations for that code may be scattered around the TC. On SPARC, the fast branch form covers a range of 8MB. Thus branches within the TC can use the fast form if the TC is less than 8MB. Branches to and from the TC (e.g. to and from the simulator) may be much more than 8MB away, so the long/slow form must be used.[13]

It is sometimes reasonable to preload static code fragments in the TC to ensure they are within branch distance. For example, rarely-taken full-buffer returns can use a conditional branch to nearby *trampoline code* that performs a long/slow jump to the main loop's full buffer return. For example, Figure 15a shows a short prologue. On the rare occasions that the buffer is full, it jumps to a nearby copy of the fragment in Figure 15b, which in turn returns to the main loop. Note that in many existing SPARC implementations, taken branches are less expensive than untaken branches. Thus, in the non-full common case, this shorter sequence takes longer to run than the longer sequence used currently in Shade.

```
prologue:
    subcc   %ntb, count, %ntb
    blu,a   eot_trampoline
    addcc   %ntb, count, %ntb
body:
```

Figure 15a. A shorter prologue

```
eot_trampoline:
    call eot
    nop
```

Figure 15b. Trampoline code (replicated)

Common cases should avoid trampoline code, but near branches can still be used if, e.g., the translation epilogue uses a near branch to a replicated copy of the main loop.

13. SPARC has an unconditional non-annulling fast far branch, `call`, that is actually used in most cases.

6.11. Application Traps and Signals

A *signal* causes a change in the control flow without an explicit control transfer instruction. *Asynchronous* signals arise from external events such as timer expiration and keystrokes. *Synchronous* signals, also known as *traps*, arise because of, e.g., arithmetic overflow or invalid memory references. Traps always interrupt the instruction that caused them, and can be predicted by examining the machine state before starting the instruction.

This section considers signals that are meaningful to the application, as compared to Shade or the analyzer. We distinguish between *arrival*, when the signal is delivered to Shade, and *delivery*, when the application starts executing instructions to handle the signal.

Application signals can arrive when Shade is running analyzers, the dynamic compiler, etc. In these circumstances it may be impossible to execute the signal handler immediately because, e.g., the trace buffer or TC is full.

Signals also must be delivered when the virtual machine is in a consistent state. The virtual state can be inconsistent during translation execution because Shade keeps some virtual machine state in host registers and emulates target instructions using multiple host instructions. The virtual state is generally consistent at translation boundaries. It is possible to generate code so that state is consistent on finer boundaries, but doing so interferes with optimizations.

Thus, signal delivery to optimized code can use three techniques: compiling code at lower optimization, delaying signal delivery until the state is consistent, and mapping host state back to virtual state on demand. The complexities discussed below are only needed for applications that receive signals. Other programs, including those that ignore or "default" signals, run at full speed.

6.11.1. Asynchronous Signals

Asynchronous signals can be delayed and delivered at a convenient point, when resources (e.g. empty trace buffers) are available and the virtual state is consistent. Each signal arrival causes invocation of a host signal handler that queues the signal and resumes execution of the simulator. At some convenient point, such as execution of the simulator main loop, the global signal queue is checked. If there are signals pending they are handled instead of running the regular application code.

Translation chaining may create a cycle that executes indefinitely, making it impossible to check for signals reliably in the main loop. The chaining cycle can delay signal delivery until program termination, or even forever if the loop is waiting for arrival of a signal. Potential solutions include:

- Ensure chaining cannot form cycles, so that control will eventually return to the main loop. Since chaining is a useful optimization, don't turn it off completely. Instead, backward chaining (chaining to lower addresses in the TC) is turned off [DS84]. Since no cycles can form, control always returns eventually to the main loop. Turning off forward chaining can leave cycles for self-chained translations.
- Allow chaining to form cycles and put a signal check in each translation. That makes each translation larger. As an optimization, forward-chained translations can skip the flag check.
- Translations can be unchained on signal arrival, so cycles are broken by signal delivery.

6.11.2. Signal Delivery by Unchaining

Unchaining requires a mechanism to find the chaining slots in each translation. For example, each translation can be marked in a table that describes the start and chain slot(s).

The simplest solution is to unchain all translations on signal arrival. *Selective unchaining* reduces the delivery cost by unchaining only the translation that is executing when the signal arrives. The host signal handler examines Shade's call chain, starting with the program counter at the time of signal delivery. If the current host program counter arises from a translation, that one translation is unchained. Otherwise the translations will eventually be reinvoked via the main loop, which will test for signal delivery, so no unchaining is needed.

With selective unchaining and signal detection in the main loop, a race condition occurs when a signal arrives after the main loop signal test and before the next translation is invoked. The main loop test indicates there is no signal, but the host program counter does not arise from the TC, so no translations are unchained. Thus, the host signal handler (which defers delivery) must treat certain host program counter values in the main loop specially, branching to special code depending on the particular main loop instruction. The special code cleans up if neces-

sary and invokes the signal handler.

The main loop *fast path* finds unchainable translations that hit in the TLB first entry. The fast path is executed frequently, so a signal delivery test may be substantial addition to the main loop cost. Unchaining can be used to avoid signal testing in the main loop. When a signal is delivered to unchained translations, the return is set to a special entry point in the main loop that tests for signals. Signals that arrive during the fast path are detected by checking the host program counter to see if it is in certain parts of the main loop, as above. Slower paths through the main loop perform the normal signal test.

6.11.3. Simulating Synchronous Delivery

Traps can be predicted by examining the machine state before executing the instruction. Thus, it is always possible to prevent (simulate) traps. For example, traps arise because of a store to an unwritable page, arithmetic overflow, etc.

Some machines trap on arithmetic exceptions such as overflow. Target machine traps can be simulated with compares and branches, at a slight, extra expense.

A harder problem is that some host machines always, e.g., trap on overflow, but the target machine does not. That is, it is difficult to *avoid* a trap. Solutions include:

- Install a trap handler that simply returns without handling the exception. Traps are expensive so the slowdown may be extreme. Also, ignoring the exception may not always be what is desired, so the exception handler may need to test, or the exception handler may need to be changed periodically.
- The operation can be performed in a way that can never trap, e.g., performing a 32-bit add as a pair of adds of the 16 bit parts, and then combining the parts.
- The translation can pretest to see if the operation could cause an exception. If the operation might cause an exception, an expensive trap-free simulation is used. When a trap is never possible, faster code is used.

6.11.4. Synchronous Delivery

Simulating traps is often expensive. For example, testing page write permissions requires a test on every store instruction, even though few writes will trap. Instead of simulating traps, they can be delivered as synchronous signals to the host.

Synchronous signals must halt the current target machine instruction. Thus, if synchronous signals are taken and not simulated, it must be possible to interrupt a translation in mid-execution and clean up state so that no information is corrupted or lost. It is thus necessary to keep a *translation map* that has information used for interrupting partly-completed translations.

Synchronous signals that are delivered to the application should only arise from instructions that are directly simulating the target machine. Synchronous signals caused by, e.g., prologue and epilogue code or saving of trace data are the result of errors. Thus, the translation map needs only entries for instructions that both simulate the target machine and can cause exceptions.[14]

Thus, for each instruction that might trap, the translation map has an adjustment for the current virtual PC, trace buffer entry, and number of unused trace buffer entries. It also describes how to write back host registers to the target virtual state so the virtual state is consistent with the start of the current virtual instruction. If the target condition codes are currently in a host register, they, too, must be stored to the virtual state.

One application instruction may map to multiple host instructions. Partial results must not be stored back to the virtual system state. Generally, synchronous delivery prohibits optimizations that make it hard to map the host state back to valid (consistent) application state.

Synchronous signals should not happen in code outside of the TC, with the exception that signals can arise from a small set of utility functions, such as code that simulates multiply and divide. If the utility functions update state in well-defined ways, it is sufficient to search the host call chain, restore spilled registers, and replace the current PC with the PC of the translation instruction that called the utility function. Then, signal handling can

14. In some implementations, other instructions could trap and some of those need translation map entries. For instance, instructions that save trace data could trap to signal trace buffer overflow.

proceed as if the signal happened while the PC was in the TC.

Parts of the translation map can be recreated on demand by re-running the compiler. However, care is needed in the presence of self-modifying code.

Note that the analyzer may want a trace buffer entry for the instruction that caused the signal, even though the virtual program counter is not advanced past that instruction.

6.11.5. Condition codes

Condition codes are allocated across target machine blocks, and therefore across translations. However, the condition code register does not always hold the virtual machine condition codes.

One problem is that the virtual condition codes are discarded when it is expected they will be set before they are used again. That omits an expensive save back to the virtual state. However, it also leaves the virtual target's condition codes inconsistent with the condition codes of the real target. Making sure the target condition codes can always be found in the virtual state requires deoptimization of the translations so that condition codes are always saved at translation boundaries. Fortunately, asynchronous signals rarely examine the condition codes exactly because the signal arrives asynchronously, when the condition codes are in an unknown state. Therefore, the major use of the condition codes is to save and restore them, and it is usually acceptable to save and restore inconsistent codes: the condition codes are inconsistent exactly because the application will set them again before using them.

When synchronous signals are delivered, the condition codes could be in a host register or they could be in the virtual state. Thus, the host signal handler must save the condition codes, but to someplace other than the virtual state. Then, the handler examines the translation map to determine whether the just-saved condition codes need to be moved to the virtual state.

6.12. Detecting Self-Modifying Code

Self-modifying code presents a problem because Shade caches code, and the cached copies must be updated when the original is changed. Self-modifying code is common enough that it must be dealt with correctly [Keppel91, KEH91].

Read-only pages can be compiled and executed blindly. There are various strategies for detecting changes to writable pages:

- The SPARC Application Binary Interface (ABI) defines an interface for the updates. ABI-compliant SPARC applications use the `flush` instruction to signal changes. Most systems define equivalent instruction cache flushing primitives. When the application signals an update, the corresponding translations are flushed.
- Pages are marked as read/write data pages or executable code pages. When code is compiled from a data page, the page is marked as executable. A write to an executable page invalidates all corresponding translations and marks it as a data page.
- Translations are compiled, executed, and then discarded instead of being cached. Writable pages execute slowly because translation caching is turned off and instructions must be recompiled each time they are executed. In addition, each translation can be only as many target instructions as the target processor's instruction prefetch, typically only a few instructions.

By default, Shade uses `flush` for instructions inside the target's code segment and discards translations for instructions outside the code segment.

6.13. TC Freeing

Shade's freeing strategy—flushing the TC when it fills—is simple and usually works well. In some situations the strategy performs poorly because it discards both useful and useless translations. Chaining makes other freeing strategies hard to implement because several predecessors may chain directly to a translation that we would like to free.

When little or no tracing is being done, translations are small and control transfers happen frequently, so chaining is an important optimization. When a lot of tracing information is being saved, translations are large so space demands on the TC are high, and since translations are large, transfers between translations are less frequent, so chaining is less important. It is, therefore, sometimes acceptable to turn off or reduce the degree of chaining to allow freeing on a finer granularity.

If translation chaining is turned off, the TC can be maintained as a heap. When space is needed and the free list is empty, more space is created by selectively freeing translations. One strategy is to free them in the same order they were created (FIFO). With FIFO freeing, it is generally necessary to keep the target address with the translation so that TLB entries can be removed without searching the TLB. Another strategy is to approximate LRU by deleting translations that have migrated to the end of the TLB search list.

With restricted chaining, FIFO freeing can still be implemented. The low part of the TC is "new" space where newly-compiled translations are put. The high part of the TC is "old" space which is gradually being freed and converted into new space. The key observation is that predecessors can be chained safely whenever they will be freed before their successors. Thus, backwards chaining (jumps from high TC addresses to low) is allowed only when it goes from the old space to the new space, and forwards chaining (low to high) is disallowed if it would cross from the new space to the old space.[15]

A third option is to limit chaining so that a translation has at most one predecessor and a pointer to that predecessor. When a predecessor chains to a successor, the successor's old predecessor is unchained and the predecessor field is set to point to the new predecessor (or, rather, to the predecessor's chaining slot). Freeing a successor simply unchains the most recent predecessor, pointed to by the successors' predecessor field. Since many translations have more than one predecessor, this "one-bit reverse branch prediction" strategy can lead to thrashing. However, since most translations have only a few predecessors, remembering two or three predecessors can eliminate most thrashing.

Translations that "fall off" of the TLB can be freed speculatively or marked as no longer accessible.

6.14. Reproducible Bugs

It would be nice to have the simulator not only execute correct code correctly but also execute incorrect code "without any surprises," that is, without behaving differently than it would on a real machine. However, it is harder to run buggy programs the same as real hardware than it is to get correct programs to run correctly. For example, it may be that a certain location on the real machine always reads as a null-terminated string when used by a buggy program, but that same location is not accessible on the host machine. The address translation tricks described above can make the address space of the virtual target machine to appear identical to those of the real target machine, but at a penalty in performance. Major parts of Shade and the analyzers could be unmapped during simulation, but again at a performance penalty. Since correct programs never reference illegal locations, such costly robustness is of no use to them.

6.15. Timing

The virtual target machine does not run the same speed as the real target machine. There are two problems. First, the virtual machine runs at a different average speed. Second, the *variability* is great—a given code fragment may run quickly one time and slowly another. Most applications are timing-independent because they run under multiuser operating systems where the machine effectively runs slower when heavily loaded and the average speed changes with changing workload. Thus, the major timing problem is that simulation is slower than one would like.

Timing often depends substantially on the host and target architectures and on the host implementation. It also depends on the application's use of the target, which is in turn affected by details of the application, compiler optimizations, and so on [Magnusson93].

6.16. Level Of Detail

As described above, Shade simulates only the user-mode SPARC architectural features. It is possible to simulate more of the machine [Bedichek90a], but the execution is typically slower because the virtual-to-host mapping is degraded.

For example, if Shade simulates multiple address spaces, all loads and stores must use address translation. Translation is needed because a given virtual address (virtual in the paging sense) can translate to different physical addresses depending on the context. In general, substantial parts of address translation must be simulated. With a simulator that performs only user-level operations, all protection checks are performed by the host machine. With

15. This style of chaining executes the main loop periodically, so a main-loop signal test will deliver signals reliably.

both user and kernel operations, many operations must be protection-checked by the simulator.

Shade may touch memory in a different order or a different number of times than would be touched by a real target machine. For example, Shade will read a string of several instructions before executing the first of them. Since Shade caches translations it may also execute an instruction many times without reading it from the target memory. Such differences may break applications that depend on references to certain pages to cause faults that perform synchronization.

6.17. ABI Conformance

The simulator must faithfully emulate the instruction set architecture. However, it must emulate other features of the environment, referred to as an *Applications Binary Interface* (ABI).

Since the simulator (as discussed) only simulates the user mode of the machine, it must emulate user calls to operating system code. Basically, that means writing an OS emulation for each OS that runs on the virtual processor architecture.

Memory layout, memory reference patterns, timing, and signal handling issues are also ABI issues.

7. Historical Perspectives

7.1. SpixTools

SpixTools [Cmelik93a] first appeared circa summer 1988. It is a set of instruction level profiling tools for SPARC applications inspired by similar tools from MIPS. `spix` (à la `pixie` [MIPS86] from MIPS) creates an instrumented copy of a program. The instrumented program, running on average 1.6 times slower than the original, counts the execution of basic blocks as the program runs, and writes out these counts as the program terminates. From these counts, `spixstats` (à la MIPS' `pixstats`) may be used to print dynamic instruction profile information. Other tools include a disassembler which annotates instructions with instruction execution counts, and a program which annotates source code with statement or instruction execution counts.

7.2. Span

Address traces could not (and still cannot) be produced with `spix` as they can with `pixie`. `span` was developed in the fall of 1988 to fill this need.[16]

`span` instrumented an application, arranging for it to collect trace information as it ran, and periodically pass this information to user analysis functions which were compiled into the application. With null analysis functions, instrumented integer applications ran about 24 times slower and floating point applications about 18 times slower than native.

`span` trace information included instruction addresses, effective memory addresses, branch taken flags, and annulled instruction flags. Instruction text and decoded opcode values were optionally provided in tables for use by the analyzer, but only static code was supported. Register values used in instructions were not available. Tracing could be enabled or disabled according to instruction address, or gross opcode groupings. There was no user trace function provision.

In addition to the conflicts of interest Shade presents, `span` presented additional conflicts. Compiling analysis routines into applications was a nuisance, and, for some applications, impossible for lack of source code. Also, the analyzer could not use functions used by the application, at least not the same copy, because these functions would be statically instrumented, then traced at run time.

7.3. Shadow

Shadow [Hsu89] first appeared in summer 1989. It uses simple dynamic compilation, which makes it possible both to separate the compilation of application and analyzer, and to run dynamically linked code. These capabilities led to `span`'s rapid abandonment, despite its speed advantage. Under Shadow, with no analysis, applications run on average 64 times slower than native.

---

16. `span`'s limited lifetime limited its robustness. After resurrection it could only handle half of SPEC89: *008.espresso*, *015.doduc*, *030.matrix300*, *042.fpppp*, and *047.tomcatv*.

Shadow compiles without caching, emulates, and traces application instructions one at a time, and periodically passes the buffered trace information to a user supplied analyzer. Shadow provides the same trace information as `span`, plus the potentially variable instruction text. Shadow variants have also been developed in which integer and floating point registers are traced. Shadow has no user trace function provision.

Shadow analyzers are linked to run at the high end of memory so that the application can be loaded in its native location. This eliminates the need to perform application address translation.

Shadow supports a native execution mode (entered and exited upon receipt of a signal from the user) in which the application code is run directly, and hence, at full speed.

7.4. Shade

Shade, first appearing circa fall 1990, is a new and typically improved version of Shadow. The introduction of the translation cache more than bought back the speed advantage of `span`'s static translation.

Other improvements include more trace control and capabilities, more control over running applications, and mitigated conflicts of interest.

Application address translation is a recent addition to Shade. So far it has only been added to Shade-V8.V8 and Shade-V9.V8. Although it complicates the Shade implementation, and slightly slows down simulation, it greatly simplifies the development and use of Shade analyzers.

Prior to this, Shade analyzers resided in the high end of memory like Shadow analyzers. Shade analyzers were run by running a program (called `shade`) which loaded the analyzer at the high end of memory and then transferred control to it. This complicated compiling and running analyzers. More importantly, software development tools such as debuggers and profilers were almost useless, since they would operate on `shade`, not the analyzer, and `shade` would soon be overwritten by an application program.

Earlier versions of Shade actually had *better* performance: SPEC 89 floating-point programs ran about two times slower and integer programs about five times slower. Adding features such as address translation and support for signals reduced performance to the current levels. Although techniques such as lazy unchaining could improve performance, the added complexity isn't worthwhile, given the ways Shade is normally used.

8. Related Work

In this section we describe related work. Since Shade is both a simulator and a tracing tool, we consider instruction-set simulators, fast virtual machines, and other instruction-level tracing tools.

8.1. SPIM

SPIM is an R2000/R3000 simulator that includes some OS emulation and debugging facilities [HP93]. SPIM is written entirely in C and runs on HP-PA (Snake), 80386, MIPS, 68000, and SPARC hosts. The entire target application is decoded to IR once, when it is read in. The main simulator loop uses straightforward dispatch and interpretation. SPIM is designed for portability and has not been optimized extensively. It runs decoded instructions about 25 times slower than hardware. The time for decoding is not given.

8.2. g88

`g88` is a simulator for a Motorola 88000 processor, including privileged instructions and address translation [Bedichek90a]. The simulator interprets 88000 instructions using threaded interpreter techniques. The 88000 machine code is dynamically translated, one target instruction at a time, into interpreter code that is then cached. Threaded code is four times the size of the target 88000 code. `g88` will run ABI-compliant self-modifying code.

`g88` runs on 68000, 88000, and SPARC hosts. On the 68000 host, each target (simulated) instruction takes an average of 20 host instructions. Figures should be similar for other hosts because the simulator does not try to exploit complex host instructions [Bedichek90b].

The simulator uses the host machine types and bit-level representations, rather than simulating those of the 88000. Operations are performed in the host's native mode and without conversion to and from the 88000 representation, but programs that rely, e.g., on the bit-level representation of floating-point values will not run on the simulator when the simulator is run on a machine that uses a different bit layout for floating-point types.

The 88000 simulates address translation for multiple address spaces and kernel-mode instructions. All application memory references are translated and protection-checked, so bugs in the application program being debugged cannot cause the simulator to fail. `g88` has been used to debug and run operating system kernels before the real hardware was available. It is also used to debug code that is difficult to debug on the real hardware.

`g88` has also been extended to simulate multiprocessors. One version uses a different decoded instruction format to reduce the number of host memory references needed to simulate a target instruction [Magnusson93]. Simulation takes typically 35-40 instructions per simulated instruction on a SPARC, and 55-60 instructions per simulated instruction on an HP-PA. Context switches between virtual processors every 10 simulated instructions almost halves performance, and context switches every cycle reduces performance to 20% of the original speed. With coarse-grained context switching, sampling to record the type of each instruction slows execution to 10% of the original speed.

Another multiprocessor version uses the original instruction format and keeps separate decoded instruction pools for each virtual processor [Bedichek93]. Using separate pools simplifies decoding and speeds context switching, but memory size increases with increasing number of processors. On a SPARC, context switching every 100 instructions requires typically 45 instructions per simulated instruction. Context switching every simulated instruction reduces performance only to about 70 instructions per simulated instruction.

8.3. Mimic

Mimic [May87] is a dynamic incremental compiler that emulates the user-level environment of an IBM S/370 on an IBM PC/RT processor. Mimic is an emulation tool, not a tracing tool. Like Shade, it assumes correct programs; buggy programs may see a machine emulation different than the real machine.

Although Mimic and Shade are similar at a high level, they differ in many details.

- Where Shade maps the target program counter to the host PC with a TLB, Mimic uses a linear table that consumes 4 bytes for each 2 bytes of target memory; there are a total of 2^{24} bytes of target memory. The linear table simplifies dispatch but is large and somewhat sparse, potentially leading to locality problems.
- In Shade, the units of translation are approximately basic blocks. Mimic does complicated inter-block analysis so that clusters of basic blocks can branch directly to each other instead of indirectly through the program counter map. Frequently, inner loops or all of small procedures can be translated as a unit. Mimic can, therefore, often use fast host branches instead of a complicated emulation of S/370 branch instructions.
- Shade compiles translations that are always correct. Mimic compiles optimistically, assuming certain values, especially the value of the base register used in jumps. Correspondingly, Mimic prologues check the assumptions. If they no longer hold, Mimic recompiles the translation, generating slower but more general code.
- Shade can have many translations for a single target instruction. Mimic has only one translation for a given target instruction. Correspondingly, Shade has one prologue and one or two epilogues for each translation. Since Mimic translates clusters of basic blocks, there may be many entry and exit points, thus many prologues and epilogues.
- The Mimic host and target machines have the same number and size of registers. Instead of doing register assignment, there is a static mapping between host and target registers. Target registers are allocated at essentially all times, notably across translations. Scratch registers are generated by temporarily spilling target registers.
- Mimic does not run programs that use self-modifying code. The lone exception is in supporting the Execute instruction, which allows code to be generated in a machine register and then executed.
- Some target machine features are not implemented: The S/370 supports unaligned multibyte accesses. Mimic does not correctly handle unaligned references; it merely continues, erroneously, and silently. Mimic does not support program exceptions. Mimic also does not support floating-point instructions. Supporting these features would change the performance in unknown ways.

When a given program is recompiled (or rewritten) for the PC/RT, it executes about two times as many instructions as on the S/370 host. Code produced by Mimic executes roughly twice as many instructions as that, or about four times as many instructions as the S/370 host. These numbers are rough but give an order-of-magnitude indicator of the good code quality.

Code generation is expensive compared to Shade; about 2,000 host instructions are required to compile a target instruction, and instructions are typically compiled twice during a single run of a program because some op-

timistic assumptions do fail, forcing recompilation. The amortized cost of analysis and code generation is not reported. Based on given static program sizes, dynamic instruction counts, and assumed instruction makeup, it appears analysis and compilation is more than 90% of the total cost of running a program. Caching compiled code across program executions is suggested to reduce the overhead.

8.4. ST-80 and SELF

ST-80 [DS84] is a high-performance virtual machine for Smalltalk-80 [GR83], and SELF is a virtual machine for a similar bytecoded virtual machine [CUL89]. Both use lazy dynamic compilation of procedures ("methods" in Smalltalk parlance) to transform bytecodes into native machine instructions. Compiled procedures are cached so that when the bytecode sequences are re-executed, the already-compiled code can be used, avoiding recompilation. Although bytecodes are designed for quick interpretation, executing native code with ST-80 makes programs run up to twice as fast as a standard interpreter. The more generic instruction set of SELF is at least two orders of magnitude slower with interpretation. The SELF compiler uses sophisticated analysis and aggressive in-line code expansion. Dynamic compilation is thus more expensive than ST-80 and the resulting code is roughly five times faster.

The performance increase over conventional interpreters comes from several sources including: (a) caching compilations to avoid retranslation (reinterpreting) costs, (b) inlining of operations known only at run time, and (c) common subexpression elimination. SELF also optimizes based on various assumptions and compiles new code when those assumptions fail.

The code generator for ST-80 has been ported to CISC and RISC architectures including the Motorola 68000, VAX, MIPS, RS/6000, and SPARC. For ST-80, dynamic compilation generally takes about fifty native machine instructions to generate each machine instruction [Deutsch90]. Data formats are not specified for the Smalltalk-80 virtual machine, so all operations can use the host machine data formats. Byte order and alignment problems are thus avoided. Execution sessions can be moved from one system to another, and internal representations are transformed when the session is restarted on the new machine.

8.5. Insignia SoftPC

SoftPC, by Insignia Solutions, simulates an Intel 80286-based IBM PC running MS-DOS. It runs on a variety of platforms based on the DEC Alpha, HP-PA, MIPS, Motorola 68000 and 88000, and SPARC. SoftPC uses dynamic compilation [Nielsen91] and allows the simulated IBM PC to execute self-modifying code [Bedichek90b]. SoftPC only simulates and does not trace. Performance figures and implementation details are not known, however it is estimated that a 68030 at 30MHz will run applications about the same speed as an 80286 at 6-8MHz [Insignia91], which indicates that it costs approximately 5-10 68030 instructions per simulated 80286 instruction.

8.6. Spa

Spa [Irlam93] is a set of tools which run on SPARC systems and are used to analyze the performance of SPARC application programs. The main component of the Spa package is a SPARC simulator and trace generator. Tools which read the traces (from a file or pipe) include SPARCstation 1 and 2 cache and pipeline simulators.

The application is placed in memory at its native location; the simulation and trace generation code reside elsewhere. Simulation itself is about 40 times slower than running native. With trace generation the speed is about 600 times slower than running native.

Self-modifying code is handled correctly. Simulation and tracing continue in both the parent and child process after a `fork` system call, and in the new program after an `exec` system call.

Application signal handlers (and, recursively, the functions they call) are directly executed instead of being simulated upon receiving a signal, so no trace information is generated. The simulator loses control during application signal handling, and so can't provide the special handling that it normally would for certain application system calls. If the application signal handler does not return normally (e.g. if it calls `exit` or `longjmp`), the simulator does not regain control of the application.

Spa and Shade use similar core simulation techniques. Spa maintains a mapping from target PC values to addresses of special-purpose simulation code fragments. Each code fragment simulates one or more target instructions. The target PC is mapped to a host PC using a lookup table (with an organization similar to `g88`'s address translation table). The current implementation does not demand-compile special-purpose simulation code but instead dispatches to a routine (written in assembly language) that does general-purpose decode and dispatch simula-

tion.

Spa, like Shade, simulates user-mode SPARC code and the SunOS 4.x operating system interface. Like Shade, it assumes the programs it runs are correct, and does not try to reproduce target machine behavior in the presence of errors.

8.7. ATUM, MPTRACE, Pixie, Titan, TRAPEDS

The analysis tools described in this section all run with identical host and target machines. None of the systems allow the tracing level to be changed dynamically, most punt on signal issues, and only ATUM traces dynamically-linked code. None of the systems allow user-supplied pre- and post-instruction functions.

Pixie [MIPS86] is a program analyzer that reads the entire executable for a program and writes a new version of the program that saves profiling information as it runs. Although the in-house Pixie is rumored to be quite powerful, the released (commercially-available) Pixie allows the user to collect only basic-block execution counts and text and data traces. There is no provision for user-defined routines to collect or analyze data. All translation from the original program to the self-tracing program is static, so the cost of translation can be amortized over multiple runs of the program. However, because all translation is static, Pixie cannot trace code that is modified at run time, so it is not possible to trace, e.g., dynamic linkers. The self-tracing program creates a data file as it runs and a separate postprocessor reconstitutes the basic block counts from the data file.

A similar system, based on link-time code modification, has been implemented on the TITAN [BKLW89]. The tracing system traces multiple processes and can perform simulation on-the-fly. Programs run about 10 times slower, not including analysis. A recent DECstation version traces both user-space and kernel-space MIPS code under single-user Mach and Ultrix [Chen93].

MPTRACE [EKKL90] collects instruction and data references for multiprocessor programs. It is similar to Pixie but annotates assembly code instead of object code, runs on a multiprocessor, and collects more trace information. Multiprocessor tracing is harder than uniprocessor tracing because timing changes can change the address trace. For example, timing changes can change lock contention, changing the time a processor spins waiting for synchronization. MPTRACE tries to minimize timing distortion. It analyzes and annotates the program's assembly code. As the traced program runs it saves important values in an *encoded trace*. The encoded trace is later postprocessed to produce the address trace. The trace is used for offline analysis, so multiple analyzers can be used on one trace. Using a single trace makes analysis results comparable. If different incremental analyzers are used, the program must be re-run and multiprocessor programs will give different traces on different runs. Multiprocessor applications typically run a factor of ten slower under MPTRACE. Most time is spent doing I/O, and all processors halted for most of the I/O time, so programs typically see a factor of two dilation. Uniprocessor programs run a factor of two slower, including I/O times.

TRAPEDS [SF89] is similar to MPTRACE, but the TRAPEDS analyzer is called incrementally at run time. Programs running under TRAPEDS on an iPSC/2 hypercube are a factor of ten or more slower, not including analysis time. TRAPEDS also runs on an Encore Multimax shared-memory multiprocessor [SJF92].

ATUM [ASH86, SA88] is an address tracing technique based on modified microcode, and it has been used on both uniprocessors and multiprocessors. In a sense, it is another example of a simulator running on the same hardware that it simulates. ATUM runs a program and produces an address trace that is saved to a data file. The analyzer is run offline. ATUM microcode simulates a cache to perform trace compression. ATUM requires a processor with microcode and programs may run as much as 20X slower, not including I/O.

8.8. Moxie, MX, VEST

Moxie [Mashey88] translates MIPS binaries into VAX binaries. The MIPS compilers were debugged by (a) compiling a source program for the VAX, (b) compiling the same program for the MIPS and converting it to a VAX binary using Moxie, and (c) comparing the output of the VAX and MIPS-to-VAX binaries. Moxie was used only for user-level programs; kernel execution was done on a separate interpreter. A "Moxied" MIPS binary runs about half as fast as the same program compiled for the VAX.

VEST and MX [SCKMR93] are tools for running VAX and MIPS binaries, respectively, on platforms based on DEC's AXP. The tools are similar in many details and will be referred to below as VEST except where the differences are important.

VEST is a binary-to-binary compiler. It reads a VAX target program and produces a host AXP program. The host program attempts to produce exactly the same results as the target. Binary translation is used because straightforward interpreters are too slow and because the AXP has no microcode and thus cannot emulate the target instruction set. The goal is to produce code that runs as fast on an AXP as it would run on a real machine using the target processor architecture.

Translation starts from known entry points and a flow graph is built following the target's flow control. The flow graph is then analyzed to remove unneeded operations, such as unnecessary condition code calculations, from the host code. VEST can translate most parts of most programs, but some jumps are difficult to resolve and self-modifying code simply "isn't there" during binary translation so it cannot be resolved.

Most PC-relative and absolute jumps are translated into jumps from host code to host code. Register indirect jumps and jumps to statically writable pages (writable at the time VEST is run) are handled by translating the target address at run time.

Some code cannot be analyzed by VEST, so the executable produced by VEST also includes the original target machine code, and the program is linked with a VAX interpreter. The target code is protected so it cannot be executed. Thus, jumps to the target code (such as a callback from a library function) cause a trap and invocation of the interpreter. The interpreter returns to the host code when it encounters a branch to a target fragment for which it has a corresponding host fragment.

Most target instructions can be implemented in a straightforward way with one or more AXP instructions. VEST provides the user with cost/accuracy switches. For example, the user can select either slow and precise VAX D-float emulation or fast but slightly less precise emulation using host execution. Similarly, the user can assert that the target program never makes unaligned references, which eliminates code to emulate unaligned loads and stores. Some instruction sequences need special handling to ensure atomicity. For example, the AXP does not address to the byte, so multiprocessor writes to adjacent bytes require synchronization.

Signals are generally delivered at arrival. VEST translation can be arranged to ensure target instruction atomicity, although the code is less optimized. The signal delivery mechanism examines the host program counter at the time of delivery. Atomicity of the target instructions is ensured by advancing or backing up the host program counter (if needed) to a target instruction boundary.

Some machine features are not fully emulated. For example, the processor status register (PSL) is is not available within exception handlers. The VAX page size is not emulated exactly. Big-endian MIPS programs (from non-DEC environments) are not supported.

VAX condition codes are calculated either as positive/negative/zero or, when the application demands, they are calculated completely.

Host register allocation is simple for the VAX target because the AXP has more registers than a VAX; for the MIPS, some registers (such as argument registers) are permanently assigned to AXP registers, while others are assigned based on their (static) frequency. The remaining MIPS registers are assigned to memory slots and are loaded and stored on demand.

Dynamically-linked libraries are written in host code and require a "jacket" to convert between the various target and host calling and layout conventions. Function pointers that are passed to dynamically-linked libraries need a similar jacket to perform conversions when host code calls back target code. When a host-code library generates an exception, the stack walk winds up in the VEST control loop's (host code) stack frame and the handler for that stack frame takes care of walking the (virtual) VAX stack frames.

SPEC benchmarks compiled for a VAX and converted with VEST are about 20% faster on a 167 MHz DEC 7000 than if they are run native on an 83 MHz VAX 6610. SPEC benchmarks that are compiled and run native on the AXP are about three times faster than when they are compiled and run native on the VAX. SPEC benchmarks compiled for the MIPS and converted with MX typically run two times faster on a 150 MHz DEC 3000 AXP Model 500 than when run native on a 40MHz DECstation 5000/240. Run native on the AXP, they are typically three times faster than when they are compiled and run native on the MIPS.

Most VAX instructions have multiple side-effects: both registers and memory may be modified, and some condition codes may be modified while others may be left set as the result of previous instructions. This makes Shade-like dynamic compilation less attractive because the compiler must perform more analysis, making compilation a larger percentage of the run time. Also, online flow analysis produces worse information than offline analysis

because online analysis must "cut corners" to be quick. Programs produced with VEST need to interpret target instructions only for dynamic linking of target code, SMC, and callbacks. Since VEST performs no tracing, there is no need for dynamically-adjusted trace collection.

8.9. Direct Execution by Disassembly and Recompilation

Shade incrementally translates target machine code to host machine code, then simulates by executing the host code. An alternative is to perform an offline decompilation of each target instruction to some high level language, instrument the decompiled code, then compile the high-level language [FC88].

Portability is excellent because the host machine can be any machine with a compiler for the high-level language. The compiler can also optimize simulation and instrumentation code across several target machine instructions. However, some operations are simulated in detail even on machines that support direct execution because the optimizer is unable to map decompiled constructs to the available host operations. Condition codes are a particularly difficult and expensive example.

Decompiling machine code is at best difficult because it is hard to tell what is code and what is data [May87], so the technique is implemented as decompilation of assembly code. The technique can translate groups of target instructions to improve efficiency; the reported implementation simulates instructions individually. Because the technique relies on static compilation, self-modifying code cannot be traced. Likewise, the tracing level cannot be changed dynamically. Exception and signal issues are not discussed. Simulation via disassembly and recompilation was 6-8 times slower than native execution.

8.10. Host-Code Multiprocessor Simulators

Tango Lite [GH92] (a derivative of Tango [DGH91]) and PROTEUS [BDCW91] are multiprocessor simulators that simulate the timing and behavior of shared (interprocessor) events. Programs are compiled for and executed on the host. Operations are simulated if they are missing from the host. Tango augments assembly code with simulation code. In PROTEUS, augmentation is performed by the compiler. Target timing and memory reference behavior are derived from the *host* code, by assigning a target cost to each host instruction. This assumes the host and target are "close enough." However, since these simulators are used to study the general behavior of cache and network protocols, the inaccuracies are typically tolerable.

Tango Lite and PROTEUS are designed to run programs on nearly-identical hosts and targets. The greater the host/target discrepancy, the greater the "coloring" of the simulation. Tango Lite and PROTEUS do not work on arbitrary programs and do not support general self-modifying code (though in principal they could support dynamic linking of instrumented libraries). Tango Lite and PROTEUS trace memory references; the tracing level can be changed by setting flags for the functions that simulate memory references. The tracing of signals and exception handlers is not discussed.

Per-simulated processor performance varies widely depending on the application, the number of target processors, and on the detail of simulation. Tango Lite shows slowdowns of 10 to 750. PROTEUS shows slowdowns of 2 to 500. Most overhead is from the simulation of the caches and network and in context switching between simulated processors.

8.11. Wisconsin Wind Tunnel, Unimplemented Instruction Libraries

In the Wisconsin Wind Tunnel (WWT) [HLRW92, RHLLLW92], programs are compiled to native code for the SPARC processors of the Connection Machine CM-5. References to shared memory use fine-grained distributed shared virtual memory [LH89]. Each 32 bytes of CM-5 memory has error correcting codes that can be set to cause a trap on reference. WWT implements distributed shared memory two ways. When a page resides entirely on one node, that node has page access permissions and other processors do not. When a page is shared by several processors, all of those processors have page access permissions. Any given 32-byte fragment in the page is *owned* by one processor; other processors set ECC (error correcting code) bits for that fragment to an "error" value. A non-owner reference to a location causes a memory error trap. The trap handler emulates the coherency mechanism, fetches the memory, and changes the ownership. The traps also generate events that help simulate the program's execution time using event-driven distributed event simulation.

Many processor families have implementations where some members do not implement some instructions. Executing a missing instruction causes a trap to an emulation routine.

Both the WWT and unimplemented instruction libraries are examples of machine simulation where, like with Shade-V8.V8, most operations run using native hardware. They demonstrate a fuzzy line between emulation, simulation, and software implementation.

9. Conclusions

Shade is a virtual machine and program profiler. It emulates a target system by dynamically cross-compiling the target machine code to run on the host. The dynamically-compiled code can optionally include profiling code that traces the execution of the application running on the virtual target machine. Profiling information passes from Shade to a user's analysis routines through memory which reduces the delay and trace size limitations imposed by I/O based profiling.

Shade improves upon its predecessors in several ways. First, Shade is fast. Shade simulates SPARC programs typically 2.3 (for FP programs) to 6.2 (for integer programs) times slower than they would run directly on SPARC hosts. Second, profiling may be dynamically controlled by the user. The less application code that Shade is asked to profile, the faster Shade runs. Third, profiling is extensible. The user has access to the virtual machine state and may collect arbitrarily detailed information about the application program as it runs. Finally, where other tools each provide a few useful features, Shade provides many of their features, all together in one tool.

10. Acknowledgements

Shade owes much to its predecessors, particularly its immediate predecessor Shadow, which was developed by Peter Hsu. Robert Cmelik developed Shade, with numerous suggestions from David Keppel. Steve Richardson, Malcolm Wing, and other members of the Shade user community provided useful user interface feedback and helped debug Shade. Robert Bedichek gave helpful comments on a draft of this paper. Finally, authors of many of the systems in the related work section went out of their way to help us understand their tools.

References

[ASH86]
Anant Agarwal, Richard L. Sites, and Mark Horowitz, ‘‘ATUM: A New Technique for Capturing Address Traces Using Microcode,’’ *Proceedings of the 13th International Symposium on Computer Architecture*, pp. 119-127, June 1986.

[BDCW91]
Eric A. Brewer, Chrysanthos N. Dellarocas, Adrian Colbrook, and William E. Weihl, ‘‘PROTEUS: A High-Performance Parallel-Architecture Simulator,’’ MIT/LCS/TR-516, Massachusetts Institute of Technology, 1991.

[BKLW89]
Anita Borg, R. E. Kessler, Georgia Lazana, and David W. Wall, ‘‘Long Address Traces from RISC Machines: Generation and Analysis,’’ DEC-WRL Research Report 89/14 (Appears in shorter form in *Proceedings of the 17th Annual Symposium on Computer Architecture*, May 1990, pp. 270-279.), September 1989.

[Bedichek90a]
Robert Bedichek, ‘‘Some Efficient Architecture Simulation Techniques,’’ *Winter 1990 USENIX Conference*, January 1990.

[Bedichek90b]
Robert Bedichek, *Personal communication*, December 1990.

[Bedichek93]
Robert Bedichek, *Personal communication*, April 1993.

[CUL89]
Craig Chambers, David Ungar, and Elgin Lee, ‘‘An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes,’’ *OOPSLA ’89 Proceedings*, pp. 49-70, 1-6 October 1989.

[Chen93]
Brad Chen, *Personal communication*, April 1993.

[Cmelik93a]
Robert F. Cmelik, ‘‘SpixTools Introduction and User’s Manual,’’ SMLI TR93-6, 1 February 1993.

[Cmelik93b]
Robert F. Cmelik, *Introduction to Shade*, Sun Microsystems Laboratories, Inc., 1 February 1993.

[Cmelik93c]
Robert F. Cmelik, *The Shade User’s Manual*, Sun Microsystems Laboratories, Inc., 1 February 1993.

[Conway58]
M. E. Conway, ‘‘A Proposal For An UNCOL,’’ *Communications of the ACM*, vol. 1, no. 10, pp. 5-8, October 1958.

[DEC81]
VAX Architecture Handbook, Digital Equipment Corporation, 1981.

[DGH91]
Helen Davis, Stephen R. Goldschmidt, and John Hennessy, ‘‘Multiprocessor Simulation and Tracing Using Tango,’’ *Proceedings of the 1991 International Conference on Parallel Processing (Vol. II, Software)*, pp. II 99-107, August 1991.

[DS84]
Peter Deutsch and Alan M. Schiffman, ‘‘Efficient Implementation of the Smalltalk-80 System,’’ *11th Annual Symposium on Principles of Programming Languages*, pp. 297-302, January 1984.

[Deutsch90]
Peter Deutsch, *Personal communication*, September 1990.

[EKKL90]
Susan J. Eggers, David R. Keppel, Eric J. Koldinger, and Henry M. Levy, ‘‘Techniques for Efficient Inline Tracing on a Shared-Memory Multiprocessor,’’ *SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, vol. 18, no. 1, pp. 37-47, May 1990.

[Evans92]
Doug Evans, *Personal communication*, December 1992.

[FC88]
Richard M. Fujimoto and William B. Campbell, ‘‘Efficient Instruction Level Simulation of Computers,’’ *Transactions of The Society for Computer Simulation*, vol. 5, no. 2, pp. 109-124, 1988.

[GH92]
Stephen R. Goldschmidt and John L. Hennessy, ‘‘The Accuracy of Trace-Driven Simulations of Multiprocessors,’’ CSL-TR-92-546, Stanford University Computer Systems Laboratory, September 1992.

[GR83]
Adele Goldberg and David Robson, *Smalltalk-80 The Language And Its Implementation*, Addison-Wesley, 1983.

[HCU91]
Urs Hölzle, Craig Chambers, and David Ungar, ‘‘Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches,’’ *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, July 1991.

[HLRW92]
Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood, ‘‘Cooperative Shared Memory: Software and Hardware Scalable Multiprocessors,’’ *ASPLOS V*, Boston, October 1992.

[HP93]
John Hennessy and David Patterson, *Computer Organization and Design: The Hardware-Software Interface*, Morgan Kaufman, 1993.

[Hsu89]
Peter Hsu, *Introduction to Shadow*, Sun Microsystems, Inc., 28 July 1989.

[IBM90]
IBM RISC System/6000™ POWERstation and POWERserver Hardware Technical Reference – General Information, IBM Corp., 1990.

[Insignia91]
Insignia Solutions, ‘‘SoftPC/Macintosh Questions and Answers,’’ Promotional Literature, March 1991.

[Irlam93]
Gordon Irlam, *Personal communication*, February 1993.

[James90]
David James, ‘‘Multiplexed Busses: The Endian Wars Continue,’’ *IEEE Micro Magazine*, pp. 9-22, June 1990.

[Johnston79]
Ronald L. Johnston, ‘‘The Dynamic Incremental Compiler of APL\3000,’’ *APL Quote Quad*, vol. 9, no. 4, pp. 82-87, Association for Computing Machinery (ACM), June 1979.

[Jouppi90]
Norman P. Jouppi, ‘‘Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers,’’ *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA-17)*, pp. 364-373, May 1990.

[KEH91]
David Keppel, Susan J. Eggers, and Robert R. Henry, ‘‘A Case for Runtime Code Generation,’’ University of Washington Comp. Sci. and Eng. Technical Report 91-11-04, November 1991.

[Kane87]
Gerry Kane, *MIPS R2000 RISC Architecture,* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[Keppel91]
David Keppel, ‘‘A Portable Interface for On-The-Fly Instruction Space Modification,’’ *Proceedings of the 1991 Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 86-95, April 1991.

[LH89]
Kai Li and Paul Hudak, ‘‘Memory Coherence in Shared Virtual Memory Systems,’’ *ACM Transactions on Computer Systems*, vol. 7, no. 4, pp. 321-359, November 1989.

[MIPS86]
MIPS, *Languages and Programmer’s Manual,* MIPS Computer Systems, Inc., 1986.

[Magnusson93]
Peter S. Magnusson, ‘‘A Design For Efficient Simulation of a Multiprocessor,’’ *MASCOTS ’93 - Proceedings of the 1993 Western Simulation Multiconference on International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, La Jolla, California, January 1993.

[Mashey88]
John Mashey, ‘‘Object-code to object-code translation,’’ USENET posting, 9 November 1988.

[May87]
Cathy May, ‘‘Mimic: A Fast S/370 Simulator,’’ *Proceedings of the ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques; SIGPLAN Notices*, vol. 22, no. 6, pp. 1-13, St. Paul, Minnesota, June 1987.

[Mitze77]
R. W. Mitze, ‘‘The 3B/PDP-11 Swabbing Problem,’’ Bell Laboratories Memorandum for File 1273-770907.01MF, 14 September 1977.

[Motorola84]
MC68020 32-bit Microprocessor User’s Manual, Motorola Corporation, 1984.

[Motorola89]
MC88100 RISC Microprocessor User’s Manual, Motorola Corporation, 1989.

[Nielsen91]
Robert D. Nielsen, ‘‘DOS on the Dock,’’ *NeXTWorld*, pp. 50-51, March/April 1991.

[RHLLLW92]
Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood, ‘‘The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers,’’ *ACM SIGMETRICS*, May 1993.

[SA88]
Richard L. Sites and Anant Agarwal, ‘‘Multiprocessor Cache Analysis Using ATUM,’’ *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 186-195, May 1988.

[SCKMR93]
Richard L. Sites, Anton Chernoff, Matthew B. Kerk, Maurice P. Marks, and Scott G. Robinson, ‘‘Binary Translation,’’ *Communications of The ACM*, vol. 36, no. 2, pp. 69-81, February 1993.

[SF89]
Craig B. Stunkel and W. Kent Fuchs, ‘‘TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation,’’ *Performance Evaluation Review (ACM)*, pp. 70-78, May 1989.

[SJF92]
Craig B. Stunkel, Bob Janssens, and W. Kent Fuchs, ‘‘Address Tracing of Parallel Systems via TRAPEDS,’’ *Microprocessors and Microsystems*, vol. 16, no. 5, pp. 249-261, 1992.

[SPARC8]
‘‘The SPARC Architecture Manual, Version Eight,’’ SPARC International, Inc., 1992.

[SPARC9]
‘‘The SPARC Architecture Manual, Version Nine,’’ SPARC International, Inc., 1992.

[SPEC]
‘‘SPEC Newsletter,’’ Standard Performance Evaluation Corporation.

[SW79]
H. J. Saal and Z. Weiss, ‘‘A Software High Performance APL Interpreter,’’ *Quote Quad*, vol. 9, no. 4, pp. 74-81, June 1979.

[Stallman87]
Richard M. Stallman, *GDB Manual*, The Free Software Foundation, Cambridge, Massachusetts, 1987.

[SunOS4]
SunOS Reference Manual, Sun Microsystems, Inc., 27 March 1990.

[SunOS5]
SunOS 5.0 Reference Manual, SunSoft, Inc., June 1992.

[UMIPSV]
UMIPS-V Reference Manual, MIPS Computer Systems, Inc., 1990.