Efficient Cycle-Accurate Simulation of the UltraSPARC III CPU

Peter Strazdins, Bill Clarke and Andrew Over

Australian National University
sim-devel@ccnuma.anu.edu.au

Abstract

This paper presents a novel technique for cycle-accurate simulation of the Central Processing Unit (CPU) of a modern superscalar processor, the UltraSPARC III Cu processor. The technique is based on adding a module to an existing fetch-decode-execute style of CPU simulator, rather than the traditional method of fully modelling the CPU microarchitecture. It is also suitable for accurate SMP modelling. The main functions of the module are the simulation of instruction grouping, register interlocks and the store buffer. Its simple table-driven implementation permits easy modification for exploring microarchitectural variations. The technique results in a 40% loss of simulation speed, instead of a 10 times or greater performance loss by fully implementing the detailed micro-architecture. The technique is validated against an actual UltraSPARC III Cu processor, and achieves high levels of accuracy over a range of scientific benchmarks.

1 Introduction

Architectural performance analysis is an increasingly important technique in modern computer systems design (Bose & Conbte 1998). Its main component is called *simulation*, where a model of the system is made; usually this model can reproduce the functional and, in the case of what is termed *execution-driven simulation*, the intended timing behaviour of the system. For *symmetric multiprocessors* (SMPs), the timing accuracy of the simulation is particularly important because the relative timing of events on the different processors affects program behavior as well as performance.

Thus, in order to to perform detailed simulation of a single CPU, or of an SMP system, accurate modelling of the timing characteristics of the CPU is required. This means that the simulated CPU's notion of time, as represented by its clock, must accurately reflect that of the actual processor being simulated.

An example of a project requiring detailed SMP system simulation is in the CC-NUMA Project (Australian National University n.d.). Here accurate performance evaluation of threaded computational chemistry applications is required. As such applications are memory-intensive, detailed memory system simulation (including NUMA effects), down to the explicit modelling of pipelined shared memory transactions, is performed. If the agent injecting events into the memory system (in this case, the simulated CPU) is not accurate, the timing accuracy of the simulated memory system would be wasted.

Traditionally, cycle-accurate simulation has involved full modelling of the microarchitecture. In a modern, post-RISC CPU, this involves explicit modelling of all stages of the main execution pipeline, that of any sub-pipelines, instruction grouping and (possibly) reordering, and branch prediction logic, together with the resolution of any dependencies between instructions. While this offers potentially the most accurate model, it is substantially more complex to implement and results in a 10 × or more loss of simulation speed (Pai, Ranganathan & Adve 1997, Rosenblum, Bugnion, Devine & Herrod 1997), as compared with the fetch-decode-execute style of CPU simulator.

However, it is possible to accurately predict performance, in terms of the number of clock cycles required for the execution of a sequence of machine instructions, using techniques which only model as much as the CPU's microarchitecture as is necessary for this purpose. In particular, the resource allocation of the functional units, the register interlocks, and the store buffer must be modelled.

For this purpose, 100% accuracy is not necessary; approximations and simplifications can be tolerated, for the sake of performance, provided they have only a small impact on accuracy for situations of interest.

In this paper, we will show how accurate CPU simulation can be implemented as a module which can be added to an existing fetch-decode-execute simulator, called Sparc-Sulima (Clarke, Czezowski & Strazdins 2002, Over, Strazdins & Clarke 2005), of the UltraSPARC III Cu processor (Sun 2002). The implementation is table-driven, and hence can be easily changed to explore the effects of varying instruction execution characteristics.

Related work is discussed in Section 2. Section 3 gives background on relevant aspects of the UltraSPARC III Cu execution pipeline for this work. Section 4 describes the design of the cycle counting module into the framework of Sparc-Sulima, with the validation and performance of the module being given in Section 5. Possible extensions of the module are discussed in Section 6, with conclusions being given in Section 7.

2 Related Work

Related work includes a number of cycle-accurate CPU simulators which simulate the full CPU pipeline, of which the MIPS-based SimOS/MXS (Rosenblum et al. 1997), SPARC-based RSIM (Pai et al. 1997) and SimpleScalar (Burger & Austin 1997) are well known examples. The cost of fully pipelined cycle-accurate simulation in these cases are slowdowns of the order of a thousand in the case of SimpleScalar, and several thousand in the case of SimOS/MXS, as compared

with several hundred for fetch-decode-execution simulators such as SimOS/Mipsy (Rosenblum et al. 1997).

An interesting approach to speed up cycle accurate CPU simulation is the technique of memoization in conjunction with direct execution (Schnarr & Larus 1998). The resulting simulator, FastSim, is reported to be approximately 10 × faster than SimpleScalar (Schnarr & Larus 1998). While this is an impressive result, there are several drawbacks to this approach. Firstly, it is a very complex technique that must be added to an already complex (fully pipelined) simulator[1]. Secondly, for memoization to be fully effective, a host processor with a very large memory is required. Thirdly, and most importantly for our purposes, these techniques would lose accuracy if applied in an SMP context. This is because both direct execution and memoization result in the effective skipping of the simulation over a number of cycles. While this permits large speedups to be achieved, it means that accurate interleavings of memory events is not possible.

While such simulators are termed *cycle accurate*, most studies on simulator accuracy compare simulators, or equivalent techniques, with each other, and there are very few studies that actually validate these simulators against real architectures. Indeed, in the case of RSIM and FastSim, the simulator executes a SPARC-like instruction set with a MIPS-based micro-architecture and thus cannot even be calibrated against any existing system. A study on SimOS/MXS configured for the FLASH system against the actual FLASH system has revealed that a cycle accurate simulator may not be accurate at all unless it is validated and debugged against an actual existing system (Gibson, Kunz, Ofelt, Horowitz, Hennessy & Heinrich 2000). The Talisman project is another notable study on simulator validation (Bedichek 1995).

A somewhat similar approach to ours has been made with the Sam CMT Simulator kit (Nussbaum, Fedorova & Small 2004), a series of modules which can be plugged into the SimICS simulator (Virtutech n.d.) in order to simulate UltraSPARC-based chip-multithreaded processors such as the Niagra. While simulation of chip-multithreading seems to be of principal interest, it claims to permit accurate CPU simulation by the means of an *instruction timer* module, which can vary the latency of a particular instruction type. As its purpose is to simulate the Niagra (an 8-core CMP UltraSPARC processor), the module does not model instruction grouping and register dependency effects.

The most similar work to ours we know of is (Loh 2001), where a CPU timing model is based on a general timestamping method for CPU resources (registers and pipelines) was incorporated into a SimpleScalar simulator. The model assumed in-order execution; compared with a an out-of-order model using full pipeline simulation, it was shown to be over two times faster and had an average accuracy (relative to the out-of-order model) of 5% on the SPECint95 benchmarks. However, the SimpleScalar simulator did not model any real architecture. Our work differs in its flexible table-driven design, that it models most of a real CPU including a complex set of grouping rules of and that it has been validated against hardware.

3 UltraSPARC III Cu Instruction Execution Characteristics

The UltraSPARC III Cu is a 4-way superscalar processor, in which instructions are dispatched, but do not necessarily complete, in program order. Its functional units include 2 ALU pipelines (for the execution of simple integer instructions), a floating-point/graphics multiply (FGM) and add pipeline (FGA), a branch prediction pipeline (BP) and a memory/special instruction (MS) pipeline. Each instruction within a group consumes at least one of the pipelines, and no two instructions can share a pipeline.

Details of this architecture, including a detailed description of features important to this work, such as prefetching, register interlocks and store buffer design, may be found from its manual (Sun 2002).

The MS pipeline is used for complex instructions such as integer multiply and instructions with variable latencies such a load instructions; in this case, the instruction is *recirculated* into the main execution pipeline until the operation completes. These instructions thus have a *blocking latency* $L_b \geq 0$ which blocks the execution of all future instructions for L_b cycles. With the exception of load and store operations, most instructions using the MS pipeline cause the breaking of an instruction group (before or after the instruction), or must execute in a group of their own.

ALU instructions have a latency of one cycle; FGM and and FGM instructions generally have latencies of 3 or 4 cycles. These *locking latencies* only delay the dispatch of future instructions using the output register operand of the current instruction. MS instructions can have a locking latency as well a blocking latency.

The UltraSPARC III Cu has 32 general purpose registers (available in the current register window). It also has 32 double precision floating point registers; the first 16 of these are also used to implement 32 single precision floating point registers.

Load instructions to floating point registers can be serviced from a special prefetch cache (P-cache) instead of the normal top-level data cache. As the P-cache is virtually indexed and tagged, a hit requires no address translation; thus a load floating point instruction can be serviced by one of the ALU pipelines, rather than the MS pipeline. This permits two floating point loads to execute within the same group. The CPU may steer such an instruction if a hit to the P-cache is predicted, i.e. the load hit the P-cache on its previous execution. However, details such as whether the CPU will *always* steer such an instruction, and what occurs on a P-cache miss, are not clear from the documentation (Sun 2002).

4 Design

This section describes the design of an efficient cycle counter module for the UltraSPARC III Cu. As noted in (Strazdins 2005), a few UltraSPARC III features such as block load and store operations are not currently modelled by the cycle counter.

4.1 Incorporation in the Fetch-Decode-Execute Loop

Sparc-Sulima simulates the execution of a program using a fetch-decode-execute 'run loop', as indicated in Figure 1. An instruction evaluation function table, indexed by the opcode, is used to perform the execute stage. To improve the speed of the call `DecodeInstr(CI)`, a cache of recently decoded instruction structures is maintained (Clarke et al. 2002). The variable `itracer` is a pointer to an instruction tracing module; when this is non-null, this module can be used to print the current instruction.

[1] As a result of this, the authors have begun work on providing automatic support for this approach (Schnarr, Hill & Larus 2001).

```
// pc = value of the Program Counter
CI = FetchInstr(pc);
di = DecodeInstr(CI);
// di is a decoded instruction structure
if (cycleCount)
  // a cycle counter module is installed
  clock += cycleCount->InstrCountCycles(
             di.opcode, &di.operands,
             clock);
else
  clock += 1;
if (itracer)
  // instruction tracing module installed
  itracer->TraceInstr(clock, pc, CI, di);
clock += ExecuteInstr(di.opcode,
           &di.operands);
```

Figure 1: Simplified body of Sparc-Sulima run loop, indicating the incorporation of a cycle counting module

```
clock:  pc:       opcode: operands

297621: 0x19298:  ldd     [%g1],%f2
297621: 0x1929c:  fmuld   %f4,%f18,%f6
297621: 0x192a0:  faddd   %f14,%f8,%f14
297621: 0x192a4:  add     %g1,%o4,%g1

297622: 0x192a8:  ldd     [%g2+%i1],%f18
297622: 0x192ac:  fmuld   %f4,%f20,%f8
297622: 0x192b0:  faddd   %f16,%f10,%f16

297623: 0x192b4:  ldd     [%g2+%i3],%f20
297623: 0x192b8:  fmuld   %f4,%f24,%f10
297623: 0x192bc:  faddd   %f2,%f12,%f2

297624: 0x192c0:  ldd     [%g2+%i0],%f24
297624: 0x192c4:  fmuld   %f4,%f26,%f12
297625: 0x192c8:  faddd   %f4,%f6,%f4

297626: 0x192cc:  ldd     [%i2],%f26
...
```

Figure 2: Instruction trace from an optimized matrix-vector multiply program, showing the effect of the cycle counter

The CPU's clock is simulated by the variable `clock`; this can get updated from a memory system stall upon instruction fetch (not shown in Figure 1) or execution. The clock is also nominally updated by $L = 1$ cycles upon executing each instruction. However, if a cycle counter module is installed (the pointer `cycleCount` is non-null), this latency L can be varied: if the current instruction is the first in the group, $L = L_b + 1$, where L_b was the blocking latency of the previous group. If it is not the first, then $L = 0$ if no dependencies arise through input register availability; otherwise $L > 0$ will be the minimum time for that dependency to be resolved.

Thus, the value of `clock` when `TraceInstr()` (and the instruction evaluation function) is called represents the time when the instruction is actually dispatched for execution. If the instruction is a load or store, this corresponds to the time the memory system first 'sees' the instruction. In this case, the subsequent call to `ExecuteInstr()` may add an extra latency to the instruction cycle.

Figure 2 shows an example instruction trace, where the cycle counter has been installed to update the processor clock. Empty lines separate instruction groups. As the code is highly optimized, a load, floating point multiply and add operation occur on almost every cycle; an exception is at `pc=0x192c8`, where a dependency on register `%f4` from the instruction at `pc=0x1929c` causes a 1 cycle delay. Note that this delays the instruction with the dependency, and all later instructions in the group (Sun 2002, section 4.4.1).

One approximation arises so far in the design: with respect to the memory system, the differences in the value of `clock` between `FetchInstr(pc)` and the instruction evaluation function would be smaller than on a real machine. However, for memory systems having top-level instruction and data caches with independent pathways, this can only make a difference when misses occur to both. In section 6.2, we will describe how this can be overcome, at the cost of introducing some complexity.

An alternative design not requiring a cycle counting module would be for each of the instruction evaluation functions to calculate L and incorporate this in their return values. While this has potential efficiencies in that much of the required information would be more readily available from these functions, this presents software engineering difficulties as many functions would have to be modified (over 300, in the case of Sparc-Sulima), and common data structures for the purpose of cycle counting would still have to be maintained by each one. Furthermore, modifying instruction execution characteristics (e.g. for future versions of UltraSPARC) would be difficult.

All latencies associated with the execution of the current instruction are accumulated into the return value of `InstrCountCycles()`. The following sections describe how these latencies may be efficiently computed.

4.2 Table-Driven Design

The following data associated with each instruction needs to be recorded for the purpose of cycle counting: the functional units (pipelines) used by the instruction, whether the instruction causes a group break (before and/or after its execution), its blocking latency (L_b), the latency for its destination register, and the types of the register operands (two source and one destination [2]).

It is natural to encode this information in tabular form. Noting the large number of instructions (opcodes) for the case of Sparc-Sulima, and observing that many of them share the same characteristics with respect to the above information, our design uses two tables: one mapping opcodes to opcode categories, and another table mapping opcode categories to the above information. While this creates an extra lookup step per instruction, this is a relatively small extra overhead compared with the total required work, and, as there are some 40 opcode categories, this reduces the overall space requirement and so the extra overhead may be offset by reduced cache misses.

However, the real advantage of this design is that it enables easy modification of instruction characteristics for the purpose of microarchitecture exploration, and to an extent makes the cycle counter executable code more microarchitecture-independent.

4.3 Instruction Grouping

The modelling of the instruction grouping by `InstrCountCycles()` amounts to deciding whether the current instruction can be included in the current group.

[2] A special 'no register' type can be used to represent a missing operand.

This involves firstly determining whether it can be included according to whether all required functional units (pipeline) resources are available. A bitmask of all functional units used so far by the current group is maintained by the cycle counter; a simple 'logical or' of this mask with the current instruction's functional units mask entry in the table determines this. The only complication is that there are two ALU pipelines. This is resolved by having a single ALU bit field in the current group's bitmask; this field is only set by the second instruction requiring an ALU pipeline, through the use of a flag that is set by the first.

Secondly, the group breaking characteristics of the current instruction are examined to determine if a break of a group should occur before and/or after the current instruction.

After a group breakage occurs, the maximum of all blocking latencies for the group (plus 1) is returned by the function.

For each instruction, typically only 11 load/store operations, 4 comparisons and 10 arithmetic/logic operations are required.

The bitmask implementation implicitly assumes each instruction locks its required functional units for at most one cycle, unless its has a blocking latency of $L_b > 0$, in which case it locks *all* functional units for another L_b cycles. This is valid for all UltraSPARC III Cu instructions except for the floating point divide and square root, which only lock the FGM pipeline for L_b cycles. This limitation could be overcome at the loss of some efficiency by using timestamps for each functional unit, in a similar way as it is done for register resources, as will be described in Section 4.5.

4.4 Prefetching and Floating Point Load Steering

As mentioned in Section 3, floating point loads can be steered to an ALU pipeline if a hit to the P-cache is predicted. This can affect the performance of floating point-intensive codes significantly. As we have recently implemented a detailed memory model for the UltraSPARC III Cu including the P-cache (Over et al. 2005), we are able to implement this feature in the cycle counter.

In our implementation, steering only happens if the MS pipeline is already used, and there is an ALU pipeline free, and a load has not been steered in this group. If a floating-point load gets steered, a flag is marked by the cycle counter. Upon evaluation of the load instruction, this flag is checked. If the flag is set, only the P-cache is checked. If the P-cache misses, a special exception is triggered which recirculates the instruction (including going through the cycle counter again). Since the current instruction group already uses the MS pipeline and an ALU pipeline with a steered load, the recirculated load will go in a new group.

4.5 Register Interlocks

Register interlocks is a term denoting the mechanism determining any extra latency cycles due to inter-instruction dependencies from register resource requirements.

Our design needs to be efficient on highly optimized floating point codes (see Figure 2). In such cases, with up to three instructions per group that may be updating a floating point register with a maximum latency of $L_{\max} = 4$ cycles, there could be as many as 12 floating point registers at any time which are *interlocked*, i.e. unavailable due to pending operations.

Under these circumstances, maintaining a list of registers currently interlocked (together with a list of timestamps when the interlocks expire) would be expensive, requiring traversal and compaction this list.

A preliminary solution, described in detail in (Strazdins 2005) was to use bitmasks to represent the interlocks; since registers can be locked for up to $L_{\max} = 4$ cycles in the future, a circular array of length $L_{\max}$ of two 64-bit bitmasks suffices. This requires some care, with the structure needing to be updated with the introduction of any latencies.

A simpler and more general solution (suitable for large $L_{\max}$) is have individual timestamps for each register; the timestamp signifies the time when the register will first become available. These timestamps are organized as an array indexed by the register file type (GPR, single and double floating point), and the index of the register within that file[3].

The cycle counter uses these timestamps as follows. Let t denote the current time (inside `InstrCountCycles()`, this will be the value of the `clock` parameter, with any latencies from the previous group added). If a source register of the current instruction has timestamp $t' > t$, t is updated to t' (thus introducing a latency of $t' - t$). The destination register's time stamp is then set to $t' + L_b + L$, where L_b is the current instruction's blocking latency and L is its locking latency.

The worst-case overhead (realized in the case of a floating point arithmetic instruction) involves approximately 10 load, 10 comparison and 2 store operations.

4.6 Store Buffer

While its documentation suggests that the UltraSPARC III Cu processor can sustain one store operation per cycle (Sun 2002), at least under ideal conditions (e.g. for computations with working sets fitting in the top-level caches and having unit stride memory access patterns), a series of store-intensive benchmarks with double precision data indicated that the maximum sustained rate was one store every three cycles. Thus, to gain accuracy for these computations, a store buffer having a draining rate of one store per $\Delta t_S = 3$ cycles was required.

Such a buffer can be efficiently modelled as follows. The values t_S, the time the buffer last emitted a store operation, and l_S, the number of pending stores in the buffer, are maintained. The maximum depth of the store buffer is $l_S^{\max} = 8$ (Sun 2002). Upon a store operation, the following is performed. If $t_S < t$, the store buffer can be brought up-to-date by draining $n_S = \lfloor \frac{t - t_S}{\Delta t_S} \rfloor$ stores from the buffer (l_S is decremented by n_S) and incrementing t_S by $n_S \Delta t_S$.

If the store buffer is full ($l_S = l_S^{\max}$), t_S is bought up to the next emission time ($t_S \leftarrow t_S + \Delta t_S$), and now a latency of $\delta = t_S - t$ is introduced (δ is added to the return value of `InstrCountCycles()`). Otherwise, the store is merely added to the buffer (by incrementing l_S), and no other action need be taken.

4.7 Branch Prediction

On the UltraSPARC III Cu, a branch misprediction occurs when a (conditional) branch instruction computes a different target address to what is predicted

[3] Treating the single and double precision registers as being in separate files is a simplification introduced here, deemed acceptable since codes using single and double data within the same registers over very short time intervals are rare. Exact modelling could be achieved by having individual timestamps for the upper and lower halves of each double precision register, at the expense of extra implementation overhead.

(the target address of its previous execution, or, if fresh to the I-cache, the initial prediction bit of the instruction). Misprediction causes a group breakage (after the delay slot instruction) and adds an extra blocking latency (8 cycles) to the group containing the branch.

The decoded instruction operands structure is used to record a branch instruction's prediction bit. When a branch instruction is passed to `InstrCountCycles()`, its operands structure's address is recorded. Whether the branch is taken or not is then recorded in the main run loop, and this information is passed to `InstrCountCycles()` with the next instruction (the branch's 'delay slot'). At this point, the cycle counter deals with a misprediction (if one occurred), and stores whether the branch was taken in the branch's operands structure for future use.

4.8 Counting CPU-Specific Events

The UltraSPARC III Cu has hardware support for counting events for the purposes of performance analysis. For the purposes of validating the simulator, it is important to implement these as well (Over et al. 2005). The cycle counter module has sufficient information to monitor CPU-related events such as counts of instruction executed, elapsed cycles, and FGM and FGA instructions executed (using the bitmasks for functional units usage in the instruction category table). Stall cycles from GPR dependencies, floating point register dependencies and a full store buffer can also be easily counted by the module.

This functionality proved important in the validation of the cycle counter.

5 Evaluation

In this section, we will evaluate the accuracy of the cycle counter module and its effect on simulation performance, using optimized numerical programs as benchmarks.

5.1 Validation

The first stage of validation is to manually examine instruction traces such as that in Figure 2 and see if the timestamps are consistent with the rules provided in the UltraSPARC III Cu manual (Sun 2002). This tests whether the module is accurate with respect to the CPU, as defined by its documentation.

However, the timing of real implementation of the CPU may be different to what is specified in its documentation (usually slower); so the next stage involves comparison of benchmark performance when run on the simulator and on the real machine. When discrepancies were detected, comparisons of performance counters for the simulator and the real hardware and analysis the disassembled code of the benchmark were used to gain some insights, and often microbenchmarks were constructed to verify the suspected cause.

Two intricacies in the UltraSPARC grouping rules took a long time to discover. The 'same group bypass' rule states that "no instruction can bypass the result to another instruction in the same group" (Sun 2002, Section 4.4.4). Since all result-producing instructions have a latency of at least 1 cycle, this was initially interpreted that the instructions were be grouped together, but the second instruction was stalled till the result was ready. This can be different to the situation where a group break is forced before the second instruction, which from microbenchmarks we believe to be the correct interpretation.

The second relates to grouping behaviour when a register dependency stall is encountered. The relevant documentation states that "the offending instruction and all younger instructions [already in the group] are recirculated (Sun 2002, Section 4.4.3). It does not specify if regrouping (with new instructions) is possible. It was initially assumed that it was not; however, this caused the cycle counter to be pessimistic by about 15% on optimized floating-point intensive codes such as matrix-vector multiply. Running the code with performance counters on the real hardware indicated that much fewer than expected stall cycles from register dependencies was occurring. Inspection of the assembly code indicated that if the grouping was such that the load instructions were scheduled at the bottom rather than at the top of the group (see Figure 2), this would separate the load with the dependent multiply by another cycle, eliminating the stall. Implementing regrouping in the cycle counter gave such a pattern, as the first multiply causing a stall (in the 24-way unrolled loop) caused the load instructions in subsequent groups in the loop to be scheduled at the bottom. With this, the cycle counter agreed closely with both the performance counters and the execution time of the real hardware.

We first present a set of mini-benchmark results from simple matrix-vector operations (the algorithm used for $y \leftarrow A^{-1}x$ was an iterative solver based on matrix-vector multiply).

The benchmark codes were compiled without prefetch instructions, but otherwise high levels of compiler optimizations were performed. The simulator was compiled in 64-bit mode under GCC 3.4.4 with optimization flags `-x02`. To minimize the effects of the memory system on simulated time, the simulator was configured with the store buffer, prefetch cache and write cache disabled.

Table1 gives results for vector operations; the computations were repeated a sufficient number of times (normally $100 - 10000\times$) for accurate timings with a high resolution timer, after both the instruction and data caches were warmed. As the execution time is thus dominated by a relatively small loop body, these can be classified as micro-benchmarks. The vector operation results are expressed in terms of MOPs, which is millions of vector elements processed per second. Note that the working set fit within the top-level caches in all cases. The results show that for store-intensive computations, modelling the store buffer substantially improved accuracy. With the store buffer, the results are always optimistic and within 33% accuracy (in most cases, well within).

A comparison of performance counters for the simulator and real hardware indicate that store-buffer (and write-cache) effects accounted for the bulk of these discrepancies. The table also indicates the effect of using a cycle-accurate memory system model (Over et al. 2005). This model detailed store-buffer and write-cache modelling and deals with issues such as read-after-write hazards, coalescing, and write-cache bandwidth. The memory model has been extensively validated using memory-intensive micro-benchmarks (Over et al. 2005); despite this, it tends to be pessimistic on these particular benchmarks. It has proved very difficult to develop a store-buffer and write-cache model that is accurate in all situations.

The cycle counter, together with cycle-accurate memory model, has also been validated over the NAS Parallel Benchmarks (NASA Advanced Supercomputing n.d.) (OMP version, using one thread). Table 2 gives comparisons for the main computation phase of the benchmarks. The average absolute error is 4.4%, the average error is -2.1% (the simulator is generally optimistic), the maximum error

computation	simulated				actual
	no CC	CC-SB	CC+SB	CC+SB'/WC	
$y \leftarrow x$ ($8 \times$ `ldd`; $8 \times$ `std`)	309	445	298	224	297
$y \leftarrow x$ ($1 \times$ `ldd`; $1 \times$ `std`)	150	299	299	224	224
$y \leftarrow 2x$	123	216	148	111	136
$y \leftarrow y + x$	170	288	288	222	254
$y \leftrightarrow x$	169	221	149	108	118
$a \leftarrow \Sigma_{i=0}^{n-1} \lvert x_i \rvert$	179	179	179	179	179
$x \leftarrow 2x$	210	446	298	224	256
$x \leftarrow 0$	497	866	296	223	292
$y \leftarrow Ax$	476	1376	1376	1376	1309
$y \leftarrow A^{-1}x$	442	1141	1158	1141	1062

Table 1: Speed in MOPs for vector-vector computations (length 4000) and in MFLOPs for vector-matrix computations (A is 64×64 and row-major) on an 900 MHz UltraSPARC III Cu (CC = cycle counter, SB = store buffer, SB'/WC = detailed store buffer and write-cache)

being 12%. These results are generally better than in Table 1, due to the overall intensity of store operations being less. The results of this section together thus indicate a high degree of accuracy of the cycle counter over a range of scientific applications, with the store-intensive mini-benchmarks exposing a particularly difficult situation in the validation of an UltraSPARC CPU and memory system.

5.2 Performance

Table 3 indicates the overheads of the cycle counter module. These measurements were based on execution time (process user time) of the simulator running the whole test program, with the indicated computation being repeated 1000 times in order to ensure other parts of the program (e.g. data initialization and result checking) made a negligible contribution. The column "no CC: CC-RIL-LS" indicates the slowdown of the simulator with a cycle counter module with register interlock checking and floating point load steering suppressed over that of the simulator with no module, whereas the column "no CC: CC-LS" indicates that with only load steering suppressed. The column "no CC: CC" indicates the slowdown of the simulator with a complete cycle counter module over that of the simulator with no module.

The column "host: no CC" indicates the slowdown of simulator program (without a cycle counter module) over the same test program being run natively on an UltraSPARC III Cu. This gives an indication of simulator efficiency. However, it is profoundly influenced by the speed (degree of instruction-level parallelism) of the computation when run natively. For example, the simulator's slowdown for a LINPACK Benchmark program with $n = 500$, which ran at 640 MFLOPs, was 544; thus the matrix-vector multiply, running at twice as many MFLOPs, has approximately twice as large a slowdown.

These results indicate that computing instruction groupings adds 10–20% overhead, and that computing register interlocking adds a further 10–20% overhead. Floating point load steering adds a variable amount of overhead, with the total cycle counter overhead being about 40%. For the LINPACK computation mentioned above, the overheads were similar to that of $y \leftarrow Ax$.

FastSim achieves slowdowns of between 150-300 on the SPEC95 Benchmarks (Schnarr & Larus 1998). In terms of CPI and the frequency of memory system events (which are expensive to simulate), the $a \leftarrow \Sigma_{i=0}^{n-1} \lvert x_i \rvert$ computation above would be the closest match to these. Taking this into account, FastSim seems to be faster by a factor of at least two. However, the results are not directly comparable since while both simulators are SPARC-based, SparcSulima simulates a much larger ISA, which adds a significant degree of overhead to the basic simulator (Clarke et al. 2002), and the FastSim slowdowns are based on a much earlier host system (a 167 MHz UltraSPARC I). As simulation is memory-intensive, it is advantageous in terms of slowdowns to use a host with a lower memory latency with respect to CPU speed.

6 Extensions to the Design

This section indicates how the design of Section 4 may be improved and extended to broader contexts.

6.1 Performance Enhancement via Caching

The *caching* of expensive computations, such as the decoding of instructions, has been an important technique in improving the performance of Sparc-Sulima (Clarke et al. 2002). Note that memoization (Schnarr & Larus 1998) is an extension of this technique. In the case of the cycle counter, one way of applying this idea would be to record the result (the accumulated latency) of the previous call to `InstrCountCycles()` with a given instruction in a lookup table. The table would be tagged and indexed by the instruction's address, with the `pc` now needing to be passed to the function as well.

This result could be re-used on the next execution of this instruction if it were known that the conditions over the last $L_{\max}$ cycles were identical on the previous execution. A sufficient condition for this would be whether the instruction groupings were identical over the last $L_{\max}$ cycles. In turn, this can be determined from whether the values of the `pc` were the same over the last $L_{\max}$ cycles (if an instruction caused a multi-cycle stall, the corresponding `pc` value would be repeatedly recorded; if multiple instructions were issued in the one cycle, the first could be used).

This information would be maintained `InstrCountCycles()` for the current instruction, and also stored in the lookup table. To save space and improve lookup speed, a 64-bit hash value (e.g. for the case $L_{\max} = 4$, the bits 17:2 of the `pc`'s) could be used.

On a hit, the lookup table's latency would be returned, without having to maintain the cycle counter's data structures for determining instruction groupings and register interlocks. The overhead should be quite small.

The difficulty arises upon the first miss, as these data structures have to be reconstructed. In principle this would be possible, provided the cycle counter also maintained a list of all parameters (clock values, opcode and decoded operand structure addresses) over

workload	bt.S	ft.S	is.S	lu.S	lu-hp.S	mg.S	sp.S
ratio host:sim.	1.00	1.07	0.88	1.00	0.98	0.90	1.00

Table 2: NAS Benchmark validation

computation	host: no CC	no CC: CC-RIL-LS	no CC: CC-LS	no CC: CC
$y \leftarrow x$	966	1.07	1.21	1.35
$y = y + x$	1435	1.03	1.22	1.42
$a \leftarrow \Sigma_{i=0}^{n-1} \lvert x_i \rvert$	590	1.19	1.36	1.39
$y \leftarrow Ax$	1757	1.17	1.27	1.41

Table 3: Slowdowns for selected computations of Table 1 indicating overall speed of simulator and added overheads due to the cycle counter module, for an 900 MHz UltraSPARC III Cu (CC = cycle counter, RIL = register interlocks)

the last $L_{\max}$ cycles, and 'replayed' these instructions in order to perform the reconstruction.

However, the fact that the cycle counter must maintain state data persisting over a considerable number of invocations makes it less amenable to the computation caching technique. Furthermore, branch and P-cache load mis-predictions (Sections 4.7 and 4.4) would raise further complications. For these reasons, and the fact that the cycle counter's overheads has been acceptable (within 50%), this has not yet been implemented.

6.2 Improving Fetch-Execute Timing Accuracy and Extension to Out-of-order Execution

As mentioned in Section 4, a minor source of inaccuracy in the current design is that the timestamps, as would be seen by the memory system, between the fetch and execute stage of a load or store instruction would appear closer than on the real machine, due to the number of pipeline stages (5 in the case of the UltraSPARC III) between these stages.

Furthermore, while UltraSPARC-specific behaviour is mainly confined to the tables, one implicit assumption is that instructions always execute in-order. This prevents the design from easily being adapted to other post-RISC processors, such as the MIPS and Alpha.

These limitations could be overcome by a cycle counter design that 'buffered' a number of instructions. Upon each invocation, it would firstly choose which instruction in its buffer is to be executed, then calculate as before its latency contribution, and then emit the instruction (corresponding `pc` value, opcode and decode operands) for the instruction evaluation function and the rest of the main run-loop.

This of course would add complexity, particularly if the reordering rules are complex, and considerable overhead, particularly if a large instructions buffer is needed. However, with this extension, a modular cycle counter design could still be used to capture CPU timing behaviour without needing to resort to full microarchitecture simulation.

6.3 Integration into a SimICS-like Simulator

SimICS is a complete-machine simulator that has been used in many places to simulate UltraSPARC-based systems (see e.g. (Nussbaum et al. 2004)). The core SimICS simulator provides 'hooks' for user-defined modules to be plugged in (Virtutech n.d.) and called upon certain events, such as the execution of each instruction.

The cycle counter design presented so far is independent of the other aspects of Sparc-Sulima, except in the decoded instruction operand structure. Its efficiency also relies heavily on having the decoding available. Thus, a 'wrapper' function would be needed to be called from SimICS instead; this function would take the un-encoded instruction as its parameter, maintain its own *instruction decode cache* (Clarke et al. 2002), and then call `InstrCountCycles()` with the result of the decodings.

7 Conclusions

The cycle counter presented above substantially improved the timing accuracy of CPU aspects of an UltraSPARC simulator with only modest implementation overhead ($< 40\%$) and complexity of implementation, compared with the traditional approach of fully modelling the CPU microarchitecture. It thus represents a 'sweet-spot' in the accuracy / performance tradeoff that is central to computer simulation design. It is suitable for coupling with cycle-accurate SMP memory system models, enabling a realistic timestamping of events passed to the simulated memory system. The two main features of its implementation, determination of instruction groups and that of stall cycles due to register dependencies, contributed approximately equally to the overhead.

Its modular design permits it to be activated only in the parts of the simulation where timing is important. Most platform-specific information is recorded in tables, permitting the easy exploration of varying timing-related characteristics of the microarchitecture in order to determine its effect on overall application performance. The algorithms and methods used can also be reasonably easily adapted to other post-RISC architectures, except that modelling out-of-order execution would require considerable extension.

While the approach we have taken is mainly applicable to in-order execution, we believe that the advent of chip-multiprocessing and the increasing concern over power consumption will favour the introduction of in-order cores, particularly for heterogeneous chip -multiprocessing (Kumar, Tullsen, Jouppi & Ranganthan 2005).

Manual inspection of the timings made by the cycle counter demonstrate that it closely implements the main documented timing characteristics of the UltraSPARC III Cu. However, in order to model store-intensive applications (even when the working set can be kept within the top-level cache), undocumented information on the store buffer had to be reverse engineered by experiments. Without validation against a real system, it would have been difficult to detect this large source of inaccuracy.

In the benchmarks of interest, the cycle counter proved highly accurate compared with the real ma-

chine. The main discrepancies were on store-intensive benchmarks where neither the cycle-counter's simple model, nor a highly sophisticated memory system model developed elsewhere, could model the real hardware to full accuracy in all situations.

A few aspects of the CPU are not currently modelled. It is expected that each new feature could add a small amount of overhead. In principle, this would broaden the range of applications over which the simulator is accurate. However, the real limiting factor in improving simulation accuracy is the lack of complete and accurate performance-related information available on the real, as opposed to the documented, version of the CPU.

The source code for the SparcSulima simulator, including the cycle counter, is available from `http://ccnuma.anu.edu.au/sulima/`.

Acknowledgements

The authors thank the Australian Research Council, Sun Microsystems Inc. and Gaussian Inc. This research has been funded under the ARC Linkage Grant LP0347178.

References

Australian National University (n.d.), 'The CC-NUMA project: Computational Chemistry on Non-Uniform Memory-access Architectures', `http://cs.anu.edu.au/CC-NUMA`. *http://cs.anu.edu.au/CC-NUMA

Bedichek, R. C. (1995), Talisman: Fast and Accurate Multicomputer Simulation, *in* 'Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modelling of Computer Systems', ACM Press, pp. 14–24.

Bose, P. & Conbte, T. M. (1998), 'Performance Analysis and its Impact on Design', *IEEE Computer* pp. 41–49.

Burger, D. & Austin, T. (1997), The SimpleScalar Tool Set, Version 2.0, Technical Report TR-1342, University of Wisconsin-Madison Computer Sciences Department.

Clarke, B., Czezowski, A. & Strazdins, P. (2002), Implementation aspects of a SPARC V9 complete machine simulator, *in* M. Oudshoorn, ed., 'Computer Science 2002', Vol. 4 of *Conferences in Research and Practice in Information Technology*, Australian Computer Society, Monash University, Melbourne, pp. 23–32.

Gibson, J., Kunz, R., Ofelt, D., Horowitz, M., Hennessy, J. & Heinrich, M. (2000), FLASH vs. (Simulated) FLASH: Closing the Simulation Loop, *in* 'Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems', ACM Press, pp. 49–58.

Kumar, R., Tullsen, D. M., Jouppi, N. P. & Ranganthan, P. (2005), 'Heterogenous Chip Multiprocessors', *IEEE Computer* **38**(11), 32–38.

Loh, G. (2001), A Time-Stamping Algorithm for Efficient Performance Estimation of Superscalar Processors, *in* 'Proceedings of the 2001 ACM SIGMETRICS Joint Internatiional Conference on Measurement and Modelling of Computer Systems', ACM Press, pp. 72–81.

NASA Advanced Supercomputing (n.d.), 'NAS Parallel Benchmarks', `http://www.nas.nasa.gov/Software/NPB/`. Version 3.1. *http://www.nas.nasa.gov/Software/NPB/

Nussbaum, D., Fedorova, A. & Small, C. (2004), An overview of the Sam CMT simulator kit, Technical Report TR-2004-133, Sun Microsystems Research Labs. *http://research.sun.com/techrep/2004/abstract-133.html

Over, A., Strazdins, P. & Clarke, B. (2005), Cycle-Accurate Memory Modelling: A Case-Study in Validation, *in* 'Proceedings of the 13th International Symposium on Modelling, Analysis and Simulation of Computer and Telcommunication Systems', IEEE, pp. 85–94.

Pai, V. S., Ranganathan, P. & Adve, S. V. (1997), RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors, *in* 'Proceedings of the Third Workshop on Computer Architecture Education'. Also appears in IEEE TCCA Newsletter, October 1997.

Rosenblum, M., Bugnion, E., Devine, S. & Herrod, S. (1997), 'Using the SimOS Machine Simulator to Study Complex Computer Systems', *ACM TOMACS Special Issue on Computer Simulation* .

Schnarr, E. C., Hill, M. D. & Larus, J. R. (2001), Facile: a language and compiler for high-performance processor simulators, *in* 'PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation', ACM Press, pp. 321–331.

Schnarr, E. & Larus, J. R. (1998), Fast out-of-order processor simulation using memoization, *in* 'Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems', ACM Press, New York, New York, pp. 283–294.

Strazdins, P. (2005), CycleCounter: an Efficient and Accurate UltraSPARC III CPU Simulation Module, Technical Report TR-CS-05-01, Department of Computer Science, Australian National University.

Sun (2002), *UltraSPARC III Cu User's Manual*.

Virtutech (n.d.), 'Simics 1.0', `http://www.simics.com/`. *http://www.simics.com/