Design and Implementation of a High Speed Microprocessor Simulator BurstScalar

Takashi Nakada Hiroshi Nakashima
Toyohashi University of Technology
E-mail: {nakada,nakasima}@para.tutics.tut.ac.jp

Abstract

This paper describes the design and implementation of our high speed simulator for out-of-order microprocessors named BurstScalar. The simulator is based on the well-known SimpleScalar simulator but its execution speed is accelerated by computation reuse technique. Each time a loop is iterated, BurstScalar consults its state transition table to examine whether the iteration turns the microarchitectural state into what has already occurred. If the behavior of the iteration matches a state transition table entry, we reuse the complicated computation for out-of-order microarchitectural simulation by simply following the transition arc registered in the table. Moreover, in order to minimize the overhead of the reuse, we apply the reuse technique only to loops with enough number of iterations. This loop selection is performed by an instruction level pre-execution which only costs 1/10 to 1/100 of out-of-order cycle accurate simulation. The evaluation of BurstScalar with SPEC CPU95 benchmarks proves its efficiency showing up to 5.1 and 2.3-fold speedups over SimpleScalar for SPECfp and SPECint respectively, and 2.6 and 1.5-fold in average.

1. Introduction

Steady progress of VLSI technology allows architects to make their microprocessors have complicated execution mechanisms. For example, out-of-order superscalars are the mainstream of desktop and laptop computers and will be as well of handhelds and embedded systems in near future. At the same time, many researchers are trying to introduce more sophisticated mechanisms such as aggressive speculation, simultaneous multi-threading, clustered superscalars and so on, to achieve higher performance and/or lower power.

For these research and development of microprocessors and systems embedding them, microprocessor simulators are indispensable to evaluate and/or to verify their functionality and performance. However, most of existing cycle accurate simulators, which give reliable performance data to architects, are 1,000 to 10,000 times as slow as real machines. Thus architects have to wait for more than *two weeks* until their simulator of slowdown 5,000 completes its work with a workload that takes only five minutes on a real computer

This inefficiency of the cycle accurate simulators is due to the complicated out-of-order instruction scheduling to be simulated in clock by clock manner. In fact, the slowdown of instruction level simulators to reproduce architectural (i.e. not *micro*-architectural) behavior is in the range of 10 to 100. Thus we may conclude that 90 to 99% of the work of cycle accurate simulators are for instruction scheduling simulation.

On the other hand, the execution of a program shows *locality* especially in loops. In a loop, a processor repeatedly executes a few (or one, often) sequences of instructions and accesses its memory and other resources in a steady manner. Thus, why cannot we expect that an instruction scheduling pattern is also repeated in a loop? If we may expect that, we can reuse the computation result for the scheduling that we once produced and skip the most time consuming part of the simulation.

Our out-of-order microprocessor simulator named *BurstScalar* is designed lead by this observation. Our contributions to microprocessor simulation technology given in this paper are as follows:

1. We apply computation reuse technique to the instruction scheduling of the most widely used cycle accurate simulator SimpleScalar[1] by dynamically building and consulting a microarchitectural state transition table.

2. To minimize the number of states in the transition table, we register only the microarchitectural states at the beginning of loop iterations, which are the most promising sources of locality and acceleration, excluding other possible execution points such as memory references and conditional branches.

Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)
1526-7539/04 $20.00 © 2004 **IEEE**

3. To minimize the overhead of the reuse operations, we apply the reuse technique only to loops with enough number of iterations which are picked by a instruction level pre-execution.

2. Related Works

Microprocessor simulators are fallen into two categories; trace driven and execution driven. As discussed in many papers, for example in [10], trace driven simulators are faster, much faster often, than execution driven ones but are less accurate because they capture only a part of processor behavior such as memory data accesses. Another source of the inaccuracy is that a trace is obtained from *logical* execution paths of a workload and thus *physical* execution paths such as speculative ones are omitted. Since our objective is to build a cycle accurate simulator, our BurstScalar is execution driven of course, but has some trace driven flavor in it. Our observation is the trace of an execution is useful to accelerate the simulation driven by the execution because we can grasp the behavior of the execution in advance. As discussed later, BurstScalar produces a kind of instruction trace in advance to pick loops and to count their iterations.

Execution driven simulators are further classified from two viewpoints; model preciseness and instruction execution mechanism. Instruction level simulators are at one end of the model preciseness spectrum where ISA-defined resources, memory and registers, are simulated in instruction by instruction manner. The other end is for cycle accurate simulators in which microarchitectural resources and mechanisms such as out-of-order pipelines, physical registers and speculation logics are simulated in clock by clock manner

Another viewpoint, instruction execution mechanism, classifies simulators into two categories; instruction emulation (IE) to interpret target machine code by software, and binary translation (BT) to execute augmented host machine code (dynamically) translated from target machine code. Roughly speaking, BT method is faster than IE method especially for instruction level simulators. For example, Shade[2] and SimOS (BT mode)[7] are well-known works in which small slowdown around 10 is achieved. Research of this category is still active in the context of retargetability and high performance techniques including IS-CS[6] proposed to achieve 3 to 4-fold speedup over sim-fast of SimpleScalar.

Although IE simulators are slower than BT ones, they are still useful especially with models more precise than instruction level. For example, detailed CPU mode of SimOS and Shaman[4] have fairly precise memory models with (coherent) caches, MMU/TLB and (shared) physical memory while their IE-type execution engines run with reasonably small slowdown, 50 to 200. Most of cycle accurate simulators including dynamic scheduling mode of SimOS, RSIM[5] and sim-outorder of SimpleScalar also employ IE method for their architectural simulation engines. It is worth to note that the slowdown of cycle accurate simulators in the range of 1,000 to 10,000 is almost insensitive to that of instruction execution about 100.

One exception of the instruction execution method for cycle accurate simulators is FastSim[8] that adopts BT method as the front-end of its instruction scheduler. Another remarkable feature of FastSim is computation reuse (or memoization in their term) for the acceleration of out-of-order instruction scheduling as we do in BurstScalar. However, BurstScalar is essentially different from FastSim in the following two technical viewpoints.

- When does a state is saved for reuse?

  FastSim tries to reuse *every* microarchitectural state transition from/to a load/store or branch instruction. Although this approach fully exploits reusability, it requires a huge number of states have to be saved blindly without any assessment of reusability. Our BurstScalar, instead, aims to reuse the state transition during iterations of loops which are most promising source of locality and thus reusability. Moreover, BurstScalar has *pre-execution* phase as its unique feature to detect loops iterating enough times for performance gain.

- How does a state is saved for reuse?

  Since FastSim saves a huge number of states for reuse, a state has to be represented as compactly as possible to keep its state store size reasonable. Therefore its microarchitectural state representation, which can be packed into 16-byte plus 1.5-byte per instruction in window, is designed for easy and efficient implementation of reuse rather than for natural and instinctive implementation of architectural idea. On the other hand, the state of BurstScalar is simply a set of data structures of SimpleScalar which are familiar to architects who use it and thus easily modified by them with their own purpose. This makes a state of BurstScalar significantly large, a few thousand bytes, but its memory consumption is reasonable because the number of states to be saved is minimized. This feature, together with IE-type execution engine based on sim-fast of SimpleScalar instead of FastSim's BT-type, makes BurstScalar much more retargetable and portable than FastSim.

Another approach of fast out-of-order detailed simulation is tradeoff of performance and accuracy. This approach is found in DirectRSIM [3] and FastILP[9]. Although these tradeoff approaches are attractive, architects might hesitate to use them because no theoretical bound of timing error is given to their own microarchitectures.

Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)
1526-7539/04 $20.00 © 2004 **IEEE**

3. Overview of BurstScalar

3.1. Conceptual Mechanism of Computation Reuse

As stated in previous sections, BurstScalar aims to accelerate the simulation of out-of-order instruction scheduling by *computation reuse* technique. Basic concept to apply computation reuse to instruction scheduling is fairly simple. Let M be a huge finite state automaton representing an instruction scheduling mechanism to be simulated, Q be the set of M's state, and Σ and Δ be the sets of its input and output symbols. If we knew M's state transition function $\delta : Q \times \Sigma^* \rightarrow Q$ and output function $\lambda : Q \times \Sigma^* \rightarrow \Delta^*$, we could easily calculate the next state of a given state $q \in Q$ with a given input sequence $a \in \Sigma^*$ by $\delta(q, a)$, and the output sequence by $\lambda(q, a)$ by simply looking up q/a pair in the state transition table of M.

Unfortunately, we do not have the transition table in advance and thus have to evaluate $\delta(q, a)$ and $\lambda(q, a)$ by simulation. However, if we fill the transition table entry when we evaluate the functions and we find q and a again, we may *reuse* the evaluation results by looking up the filled entry for q/a pair.

To minimize the overhead the table building and consulting, we may restrict the application of reuse to frequently occurred input sequences. If an input sequence a_i does not occur enough times, we simply simulate the behavior of M with its state q_i and a_i. Otherwise, the state transition table is consulted for q_i/a_i. If the entry for the pair is found, we simply set q_{i+1} to $\delta(q_i, a_i)$ and output $\lambda(q_i, a_i)$ using the record in the entry. If not found, we simulate M to have $\delta(q_i, a_i)$ and $\lambda(q_i, a_i)$ and fill the transition table entry for q_i/a_i with the evaluation results.

3.2. Definition of States, Inputs and Outputs

Now we have to map the conceptual mechanism discussed above onto a real simulator with a given machine model defining the states, inputs and outputs. A straightforward definition of a state is what the machine has in its registers/memories in the scheduling mechanism. This definition, however, takes away the chance of reuse almost completely because the registers/memories have instruction operands and results which are hardly repeated as a whole. Thus we need to remove data portion from a state to obtain certain reusability.

Fortunately, most of out-of-order simulators, including SimpleScalar which our BurstScalar is based on, separate instruction emulation and scheduling so that data structures in the scheduler have almost no data values on which instructions operate. Therefore if we transform a small exceptional data portion such as memory addresses in load/store queues into a data independent form such as access latencies, the state of scheduler is free from data values and thus much more reusable.

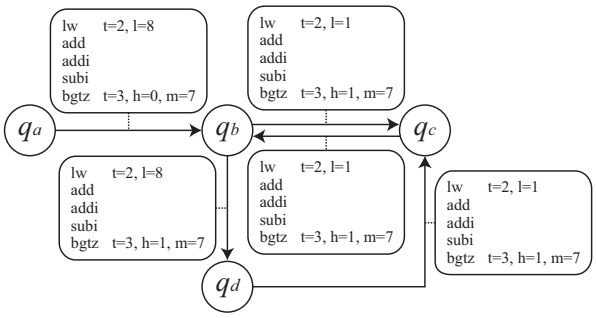

Figure 1. State Transition.

The first level approximation of the input symbol definition is to use an executed instruction as a symbol. A frequently occurred input sequence is naturally translated into an instruction sequence in a loop which iterates enough times.

However, this assumption does not hold of course because an instruction may have variable execution latency depending on its operand value or may dramatically affects the scheduling through its result. A good example of the former case is a memory access instruction whose latency depends on the cache access result and its timing. An example of the latter is a conditional branch because the scheduling of the following instructions deeply depends on the correctness of its prediction.

Thus we have to define an input symbol as an instruction optionally coupled with its data dependent behavior. In order to keep enough reusability, the coupled information cannot be the data itself on which the behavior depends but has an abstracted form representing its effect on the scheduling sufficiently. Therefore we decouple microarchitectural units which determine the data dependent behavior of instructions from the scheduler and record the interaction between the scheduler and units. The record is attached to the instruction which causes the interaction to form a input symbol.

For example, the load instruction `lw` in Figure 1 has a record indicating its access timing t relative to the base time of the instruction sequence and the access latency l. The conditional branch `bgtz` also has a record for the timing of the reference and modification of the branch prediction table (t and m) and the prediction result h. As shown in the figure, each set of recorded values for an instruction sequence may cause its own transition from a state. Therefore when we find a pair of state and instruction sequence registered in the transition table, the interaction with the units is simulated using the timing in the record and operand data value given by the instruction emulation. If the interaction results are consistent with those recorded, we successfully

Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)
1526-7539/04 $20.00 © 2004 **IEEE**

```
#define N 100          a: i=0
void foo(){            b: j=0
  for(i=0;i<N;i++){    c: body
    for(j=0;j<N:j++){  d: j++
      body             e: j<N
    }                  f: i++
  }                    g: i<N
}                      h: ret
```

(a) (b)

Figure 2. Iterations.

reuse the transition in the table.

As for the output symbols, their definition depends on what we want to know from the simulation. If we just want to measure the total execution cycles, the output function $\lambda(q_i, a_i)$ may be enough to give the number of cycles spent from q_i to $\delta(q_i, a_i)$.

3.3. Iterations

As discussed above, we need input symbol sequences that frequently occur for efficient reuse. The source of such sequences should be found in loop iterations. Thus BurstScalar at first performs an instruction level simulation to produce an instruction trace and to form it in a series of *iterations*.

In our definition, an iteration is a sequence of executed instructions bounded by backward branches, subroutine calls (branch-and-link) or an indirect branches. For example, suppose we execute a C function shown in Figure 2(a), whose control flow graph is shown in (b). The sequence of executed instructions is represented as $a(b(cde)^N fg)^N h$, where a to h are the labels in the graph and x^n is the n-times concatenation of x. Since e and g are backward conditional branches and h is an indirect branch, the sequence is transformed into the following form in which innermost parened subsequences are iterations.

$$(abcde)((cde)^{N-2}(cdefg)(bcde))^{N-1}(cde)^{N-2}(cdefgh)$$

Thus if we name the iterations as $A = abcde$, $B = cde$, $C = cdefg$, $D = bcde$ and $E = cdefgh$, the sequence is represented as an iteration trace $A(B^{N-2}CD)^{N-1}B^{N-2}E$. If $N = 100$, B occurs 9,800 times, C and D 99 times, and A and E only once in this trace.

4. Implementation of BurstScalar

Figure 3 shows the configuration of BurstScalar. As shown in the figure, BurstScalar has the following four major components.

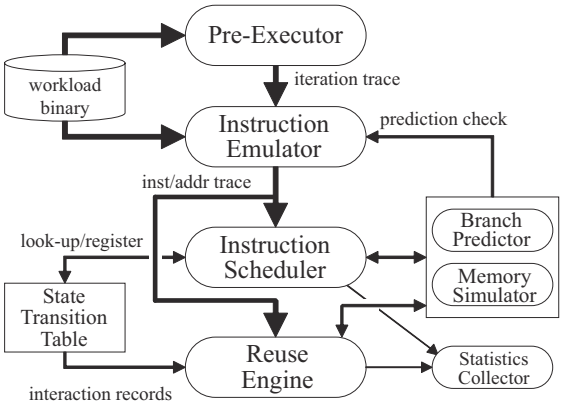

Figure 3. Configuration of BurstScalar.

- **Pre-Executor** produces the iteration trace by performing instruction level simulation on the workload.
- **Instruction Emulator** emulates workload instructions again to produce a short instruction and address traces for out-of-order scheduling simulation. It also simulates branch prediction.
- **Instruction Scheduler** simulates out-of-order scheduling. When it fetches the first instruction of a reuse candidate iteration, it looks up its state in the state transition table. If the state is found, Reuse Engine takes its place. Otherwise, it registers the state in the table and attaches the records of the interaction with Memory Simulator and Branch Predictor.
- **Reuse Engine** checks if the interaction with Memory Simulator and Branch Predictor is consistent with that recorded in the transition table. If consistent, it continues the work for the next iteration setting the scheduler state following the transition table entry. Otherwise, Instruction Scheduler takes its place.

The other component is Statics Collector which receives statistical data from Instruction Scheduler and Reuse Engine and performs collective operations on them to make statistical output of the simulation.

4.1. Pre-Executor

At first, BurstScalar invokes its Pre-Executor to execute a workload binary and to produce the iteration trace. The execution is a simple instruction level simulation based on sim-fast of SimpleScalar.

Each time Pre-Executor finds a backward branch, a subroutine call or an indirect branch, it looks up the target address in a hash table shown in Figure 4 that corresponds to the iterations in Figure 2. For example, during the inner loop of `foo` iterates, we repeatedly encounter the backward

Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)
1526-7539/04 $20.00 © 2004 **IEEE**

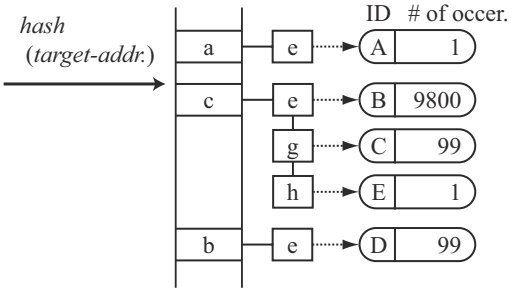

Figure 4. Identification of Iterations.

conditional branch e and find its target address c in the table. Thus we follow the link from c to find the branch e. If e is taken to iterate the loop, we put B into the iteration trace and increment its occurrence. Otherwise, the inner loop is terminated and we follow the downward link from the node for e to find the branch g for the outer loop. If branch is taken, we put C to the trace and start new iteration from its target address b.

Pre-Executor performs the operations above until the workload terminates or its on memory trace buffer of 4M entries is exhausted. In the latter case, BurstScalar's context is switched to Instruction Emulator until it consumes a half of the trace. To minimize the context switch frequency and to provide iteration counts to other components as up to date as possible, we pack the trace by a run-length encoding.

4.2. Instruction Emulator

Instruction Emulator executes the workload binary again to produce a short instruction and memory address traces given to its backend, Instruction Scheduler or Reuse Engine. Each trace is implemented as a cyclic buffer large enough with respect to the size of instruction window, and traced instruction and address are removed when the corresponding instruction retires. The fundamental job of this component is a simple instruction level simulation based on sim-fast again, but an additional job for branch prediction is assigned to it.

When it encounters a conditional or indirect branch, it suspends its execution until its backend allows to consult prediction tables. This is necessary because the Instruction Emulator cannot know when the tables are up to date for a branch by itself but the backend controls when they are modified in an out-of-order and possibly speculative manner. Then it examines the correctness of the prediction. If correct, it resumes the execution getting a new iteration from the trace if the branch is backward, indirect or for a subroutine call.

If the branch is mispredicted, it turns to false path execution mode, or speculation mode in SimpleScalar's term, for

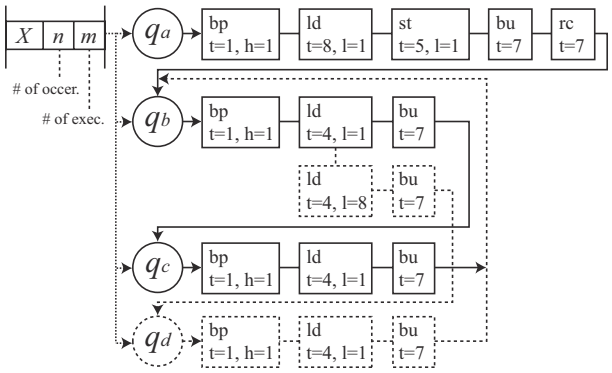

Figure 5. State Transition Table.

preserving and speculatively modifying architectural states by SimpleScalar's mechanism. Then the backend notifies the end of the false path when the branch instruction of its root reaches a pipeline stage (write-back) for the examination, Instruction Emulator recovers architectural states and resumes the correct path execution.

4.3. Instruction Scheduler

The main job of the Instruction Scheduler is what sim-outorder of SimpleScalar does. Thus the out-of-order instruction scheduling mechanism is implemented using SimpleScalar's related modules. The additional job is to build the state transition table for computation reuse.

Figure 5 shows the state transition table structure corresponds to the diagram shown in Figure 1. The root of the structure is an entry of *iteration table* which contains the number of occurrence *n* of the iteration *X* counted by Pre-Executor. It also has a counter *m* to count the execution of the iteration by Instruction Scheduler and Reuse Engine for memory management discussed later.

Each time Instruction Scheduler fetches the first instruction of an iteration, it looks up the entry of the iteration to find a state identical to its current state. If the state is found, control is transferred to Reuse Engine to skip the computation for the scheduling.

Otherwise, it checks whether the number of iteration occurrence n is greater than a threshold, set to 300 in the current implementation based on our experience. If n is less than the threshold, it simply performs instruction scheduling as SimpleScalar does because reusing will not be effective. Otherwise, the transition table entry for the current state is filled as follows.

First, SimpleScalar's data structures including Register Update Unit (RUU) and queues for instruction fetch, load/store and ALU/FPU operations are saved as the current state. A check sum is attached to the saved image so that state identity is quickly checked by eliminating hopeless candidates. Then out-of-order scheduling is per-

Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)
1526-7539/04 $20.00 © 2004 **IEEE**

formed recording the interaction with Memory Simulator and Branch Predictor. For example, after we save the state q_a in Figure 5, the first interaction 'bp' (branch prediction table lookup) at the timing $t = 3$ relative to the first fetch of the iteration is recorded together with its result $h = 0$ (misprediction). The record is linked from the state q_a and following interactions are added to the tail. In this example, a load (ld) at $t = 4$ of 8 clock latency (l), a store (st) executed because of misprediction, a branch prediction table update (bu) and the recovery of the misprediction (rc) at $t = 7$.

Then the iteration head is fetched again and thus the tail of the record list is linked to the new state q_b. After similar operations continue from q_b to q_c and q_c to q_b, Reuse Engine takes its place because q_b is found in the transition table structure. Note that at this point the dashed part of the structure for q_b to q_d and q_d to q_b is not build.

Also note that the figure is simplified omitting the followings.

- Each interaction record except for 'rc' has an offset pointer to the instruction or address trace given from Instruction Emulator so that Reuse Engine performs the same interaction with different data value.
- Interactions between Memory Simulator for instruction fetch are also recorded.

4.4. Reuse Engine

When Instruction Scheduler finds a matching state and iteration pair in the transition table, Reuse Engine starts its job. It follows the link of the interaction records from the current state and performs each recorded interaction with Memory Simulator and Branch Predictor at the recorded timing. The data value for the interaction is obtained from the instruction or address trace using the offset pointer in the record. As long as the results of interactions are identical to those recorded, Reuse Engine stays in the transition table walking along the links of states and records.

For example, when Instruction Scheduler finds its state q_b in the transition table shown in Figure 5, Reuse Engine takes its place and traverses the records of 'bp', 'ld' and 'bu' to the state q_c, then those from q_c to return to q_b, and repeats this walk until it encounters another iteration to terminate the loop. It may also stops the walk when it finds a mismatch of an interaction result and that recorded. For example, the 'ld' interaction in the path from q_b to q_c may results in cache miss with 8 clock latency.

When a mismatch is found, Reuse Engine rolls the execution back to the fetch of the current iteration head by restoring the state from what it last visited. Then Instruction Scheduler takes its place omitting interactions which Reuse Engine have already performed. In the example of the cache miss of 'ld', after the state q_b is restored, Instruction Scheduler restarts scheduling omitting 'bp' and 'ld'. Then a new record for the 'ld' of $l = 8$ is added and linked from that of $l = 1$.

After that, the interaction 'bu', the state q_d and other three records following it is created by Instruction Scheduler as explained in the previous section. Then it finds q_b and switches to Reuse Engine again. In this second invocation, Reuse Engine does not stop at the 'ld' after q_b even if the latency is not 1 but 8. Instead it follows the downward link of 'ld' of $l = 1$ and finds that of $l = 8$ and continues the walk to q_d.

4.5. Memory Management of State Transition Table

Although we try to minimize the size of the state transition table by limiting the reuse application to frequently occurring iterations, the memory area for the table may be exhausted. When exhausted, we reclaim memory space predicting the usefulness of the table contents rather than relying a simple flush or LRU scheme proposed in [8].

The prediction is based on the occurrence and execution counters n and m shown in Figure 5. If they are identical, it is worthless to keep states and records for the iteration and thus they are simply discarded. If this reclamation is not enough, we remove paths of interaction records containing cache misses or branch mispredictions for iterations of a small future usage $n - m$ expecting hitting paths are more useful than them. Further reclamation discarding all the states and the records for iterations whose $n - m$ is less than the threshold are performed if the missing path removal is still insufficient.

In order to avoid that a discarded state or path is regenerated, each iteration table entry has flags to suppress the creation until Pre-Executor is resumed and increments n to make $n - m$ enough large. The threshold of $n - m$ is doubled after a reclamation process to avoid it is invoked too frequently.

5. Performance Evaluation

5.1. Evaluation Environment

The performance of BurstScalar was measured using a 2.8 GHz Xeon based PC with 3 GB memory and Linux 2.4.20. We also measured the performance of SimpleScalar version 3.0c, which is the base of BurstScalar, for comparison. Both systems are compiled by gcc version 2.95.3 with -O2 option.

Workloads are benchmarks in SPEC CPU 95 with "train" datasets. Simulated microarchitectures are SimpleScalar's

Table 1. Simulated Microarchitecture

	model-1	model-2
I-Fetch & ILP degree		
I-fetch width	4	8
issue/commit width	4	8
RUU depth	16	256
load/store queue depth	8	128
Function Units		
integer ALU	4	8
integer mult/div	1	3
floating point ALU	4	8
floating point mult/div	1	3
Branch Prediction		
cond. branch prediction	2 K entry, 2-bit counter	
branch target buffer	2 K entry	
Caches		
L1 instruction cache	16KB, 32B block, direct map	
L1 data cache	16KB, 32B block, 4-way set assoc.	
L2 unified cache	256KB, 64B block, 4-way set assoc., 6 cycle	
memory access	$16 + 2\times$ xferred-bytes/8	

default (model-1) and extended (model-2). Those major parameters are shown in Table 1.

5.2. Iteration Occurrence

Before measuring simulation speed, we measured the number of occurrence of each iteration. Then we estimated the *significance* s_i of an iteration by $n_i \times e_i/e_T$, where e_i is the number of instructions in the iteration that occurs n_i-times and e_T is the total number of executed instructions. Thus $s_i = 0.1$ means 10% of total execution is spent in the iteration i.

Table 2 and 3 show the results obtained by summing up the significance of iterations whose occurrence are in the range of 1 to 100, 100 to 10,000, and so on. The table clearly shows that a large portion of the execution of SPECfp benchmarks is spent by frequently occurring iterations, except for 'fpppp'. It also suggests our reuse technique will be effective for them, especially for 'tomcatv', 'hydro2d' and 'mgrid', because the instruction scheduling of a frequently occurring iteration is likely reusable.

On the other hand, SPECint benchmarks tend to stay in random code sequences especially in 'go' and 'gcc'. This suggests that a large gain from reuse is hardly expected but there will still be a chance to have significant acceleration for other benchmarks.

Table 2. Distribution of Iteration Significance (SPECfp).

# of occur.	10^0-10^2	10^2-10^4	10^4-10^6	10^6-
101.tomcatv	0.0	0.3	16.0	83.7
102.swim	0.0	0.2	56.1	43.7
103.su2cor	0.0	0.1	8.9	91.0
104.hydro2d	0.0	0.5	8.4	91.1
107.mgrid	0.0	2.1	6.6	91.3
110.applu	0.2	2.6	97.2	0.0
125.turb3d	0.0	0.1	9.0	90.9
141.apsi	0.1	2.8	59.5	37.3
145.fpppp	3.0	58.7	38.3	0.0
146.wave5	0.0	0.2	28.2	71.6

(%)

Table 3. Distribution of Iteration Significance (SPECint).

# of occur.	10^0-10^2	10^2-10^4	10^4-10^6	10^6-
099.go	3.1	45.4	51.5	0.0
124.m88ksim	0.5	1.7	97.8	0.0
126.gcc	1.4	40.7	55.4	2.6
129.compress	0.2	4.7	95.1	0.0
130.li	0.1	4.0	95.9	0.0
132.ijpeg	0.1	4.0	32.3	63.6
134.perl	0.0	1.9	30.5	67.6
147.vortex	0.1	5.0	52.8	42.1

(%)

5.3. Simulation Speed

Table 4 and 5 show the simulation time of each benchmark running on BurstScalar (BS) and SimpleScalar (SS). The table also shows the speedup of BurstScalar relative to SimpleScalar (SS/BS).

As we expected from the analysis of iteration occurrence, BurstScalar greatly outperforms SimpleScalar for SPECfp benchmarks achieving up to 5.1-fold speedup for 'mgrid' (model-2) and 2.6-fold in average. Comparing the results of model-1 and model-2 gives an insight that the benchmarks are categorized in two groups by the SS/BS ratio of each model. The model-2 ratio of one group, including 'tomcatv', 'swim', 'mgrid', 'applu' and 'fpppp', is larger than that of model-1 because the model complicatedness directly affects the simulation speed of SimpleScalar while BurstScalar is less sensitive to it. That is, the more complicated the model is, the harder Instruction Scheduler have to work, but the BurstScalar skips the harder job by reuse.

As for the other group, however, the observation above is

Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04)
1526-7539/04 $20.00 © 2004 **IEEE**

Table 4. Simulation Time of BurstScalar and SimpleScalar (SPECfp).

	model-1			model-2		
	BS	SS	SS/BS	BS	SS	SS/BS
tomcatv	8016	19022	2.37	10627	26087	2.45
swim	305	1025	3.36	396	1787	4.51
su2cor	8055	24653	3.06	42428	39717	0.94
hydro2d	2524	9127	3.62	10268	12622	1.23
mgrid	4978	17074	3.43	5729	29117	5.08
applu	229	611	2.67	295	877	2.97
turb3d	6806	17791	2.61	8659	20532	2.37
apsi	1063	2824	2.66	1989	4637	2.33
fpppp	378	481	1.27	458	619	1.35
wave5	1246	3867	3.10	4016	5539	1.38
average			2.82			2.46
	(sec)	(sec)		(sec)	(sec)	

Table 5. Simulation Time of BurstScalar and SimpleScalar (SPECint).

	model-1			model-2		
	BS	SS	SS/BS	BS	SS	SS/BS
go	1150	769	0.67	1583	974	0.62
m88ksim	27	54	1.97	34	65	1.92
gcc	2613	1878	0.72	2952	3297	1.12
compress	24	46	1.92	38	61	1.62
li	138	265	1.92	243	333	1.37
ijpeg	631	1460	2.31	841	1809	2.15
perl	1857	3420	1.84	2547	3769	1.48
vortex	1771	3449	1.95	3118	3608	1.16
average			1.66			1.43
	(sec)	(sec)		(sec)	(sec)	

reversed especially for 'su2cor' which shows about 3-fold speedup with model-1 but small slowdown with model-2. The reason of the reversal is memory pressure of the state transition table. Since model-2 has a state much larger than model-1, we have to reclaim the memory space for states more frequently. Furthermore, a larger state means that an iteration has a larger number of different states and transition paths, and thus causes a significant number of state transitions cannot be reused because they were reclaimed. This observation strongly suggests that there will be a room of the improvement of the reclamation algorithm.

Another source of the necessity of the improved reclamation is found in the results of 'go' and 'gcc' in SPECint, although the performance degradation of them is in the range of our anticipation gotten from the iteration analysis. The performance of the other benchmarks in SPECint, however, is satisfactory especially that for model-1 with which BurstScalar achieves significant speedup of 1.8 to 2.3-fold.

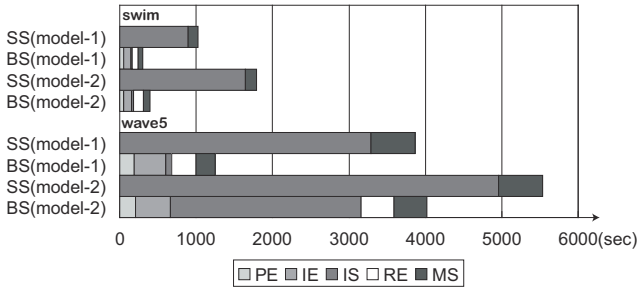

Figure 6. Simulation Time Breakdown.

Figure 6 shows the breakdown of the simulation time of 'swim' and 'wave5' run on BurstScalar (BS) and SimpleScalar(SS). The figure shows how much portion of simulation time is spent in each simulator component, which is one of Pre-Executor (PE), Instruction Emulator (IE), Instruction Scheduler (IS), Reuse Engine (RE) and Memory Simulator (ME). Note that IS of SimpleScalar includes instruction emulation because its own execution time is hardly profiled.

The graph of 'swim' clearly shows our reuse technique is quite effective and makes the execution time of Instruction Scheduler negligibly small. The portion of Pre-Executor is also small but larger than we expected. Our expectation was the instruction level simulation is as 50 to 100 times as fast as out-of-order scheduling simulation, but the result is only about 20 times. Since we confirmed that our iteration tracing consumes a really negligible execution time, this is due to a large slowdown of sim-fast which is about 100. This slowdown is much larger than those of other simulators, and thus we expect a small coding effort will improve the performance significantly. This inefficiency also makes the time spent by Instruction Emulator longer than our expectation but more essential reason should be hidden because it is almost three times as slow as Pre-Executor. Our preliminary analysis prevails inefficiency of branch prediction and false path execution which will be a good target for improvement.

As for 'wave5', it is also clear that the reuse is effective when the machine is model-1. On the other hand, the execution time of IS for model-2 is much larger than model-1. As stated before, the reason is that a significant number of states and transition paths are discarded by memory reclamation resulting much less opportunity of reuse.

5.4. Effectiveness of Pre-Execution

One important feature of BurstScalar is the Pre-Executor that finds iterations and their usefulness for reuse in advance. In order to evaluate its effectiveness, we measured the performance removing the component. Since we cannot recognize iterations, we apply reuse technique blindly at all the branches and jumps as FastSim does. Also we cannot assess the usefulness of saved states, we only perform

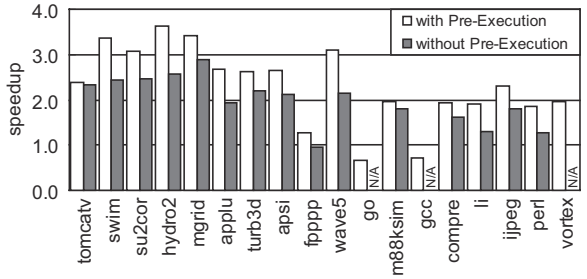

Figure 7. Speedup of BurstScalar with and without Pre-Execution.

reclamation of interaction records for cache misses and mispredictions.

Figure 7 shows the results in the form of speedup over SimpleScalar. As the graph shows, go, gcc and vortex cannot run (N/A) because they make a huge memory space for states larger than 2GB exhausted. As for other benchmarks, we see a significantly large performance degradation in the range from 8% (fpppp) to 38% (tomcatv). These results clearly show the importance of Pre-Executor not only for efficiency but also for the applicability of reuse with a reasonable size of memory space.

6. Conclusion

This paper describes our high speed simulator of out-of-order microprocessors named BurstScalar. Its most important feature is computation reuse applied to the simulation of instruction scheduling. Since the reuse technique has to be applied to the computations repeatedly performed enough times, we pick iterations that occur frequently enough in advance. Restricting the application of reuse to iterations also reduces the size of memory space for the state transition table dynamically build in simulation.

The performance evaluation results with SPEC CPU95 benchmarks prove the efficiency of BurstScalar showing a large degree of speedup over SimpleScalar, up to 5.1- and 2.3-fold for SPECfp and SPECint respectively, and 2.6- and 1.5-fold in average. The evaluation also shows the effectiveness of Pre-Executor which picks hopeful iterations for both the efficiency and applicability of the reuse technique.

Our emergent future work is to solve the performance problems revealed from the evaluation such as to improve the algorithm of memory reclamation. Other important issues including API design for easy use of our reuse technique and the extension to multiprocessor simulation are also left for future works.

Acknowledgments

This research is partly supported by a STARC research program entitled “Performance Verification System for Software Described in SpecC” and a MEXT 21st Century Research Program entitled “Intelligent Human Sensing”.

References

[1] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[2] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[3] M. Durbhakula, V. Pai, and S. Adve. Improving the Accuracy vs. Speed Tradeoff for Simulating Shared-Memory Multiprocessors with ILP Processors. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, pages 23–32, Orlando, FL, January 1999.

[4] H. Matsuo, S. Imafuku, K. Ohno, and H. Nakashima. Shaman: A distributed simulator for shared memory multiprocessors. In *Proc. 10th IEEE/ACM Intl. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 347–355, Oct. 2002.

[5] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An Execution-Driven Simulator for ILP-Based Shared-Memory Multiprocessors and Uniprocessors. In *3rd Workshop on Computer Architecture Education*, 1997.

[6] M. Reshadi, P. Mishra, and N. Dutt. Instruction set compiled simulation: A technique for fast and flexible instruction set simulation. In *Proc. 40th Conf. Design Automation*, pages 758–763, 2003.

[7] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The simos approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, 1995.

[8] E. Schnarr and J. Larus. Fast out-of-order processor simulation using memoization. In *8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 283–294, 1998.

[9] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In *ISCA*, pages 380–391, 1998.

[10] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, 1997.