# An Accurate and Efficient Simulation-Based Analysis for Worst Case Interruption Delay

Hiroshi Nakashima* Masahiro Konishi Takashi Nakada
Toyohashi University of Technology
1-1 Hibarigaoka, Tempaku, Toyohashi, 441-8580, JAPAN
{nakasima, konishi, nakada}@para.tutics.tut.ac.jp

## ABSTRACT

This paper proposes an efficient method to analyze *worst case interruption delay* (WCID) of a workload running on modern microprocessors using a cycle accurate simulator (CAS). Our method is highly accurate because it simulates all possible cases inserting an interruption just before the retirement of every instruction executed in a workload. It is also (reasonably) efficient because it takes $O(N \log N)$ time for a workload with $N$ executed instructions, instead of $O(N^2)$ of a straightforward iterative simulation of interrupted executions. The key idea for the efficiency is that a pair of executions with different interruption points has a set of durations in which they behave exactly *coherent* and thus one of simulations for the durations may be omitted. We implemented this method modifying the SimpleScalar tool set to prove it finds out WCID of workloads with five million executed instructions in reasonable time, less than 30 minutes, which would be 200–300 days by the straightforward method. We also show a parallelization of our method achieves a good speedup, about 7-fold with 8-node PC cluster.

## Categories and Subject Descriptors

C.4 [**Peformance of Systems**]: Measurement Techniques

## General Terms

Algorithms, Design, Performance

## Keywords

Worst Case Interruption Delay, Cycle Accurate Simulation

## 1. INTRODUCTION

For real-time systems and programming for them, *worst case execution time* (WCET) analysis is indispensable to assure a program or a workload completes its job with a given time constraint. Among many WCET researches (e.g. those surveyed in [11]), one of the most challenging theme is to find the upper bound of the delay caused by one (or more, sometimes) interruption occurred in the execution of a program or a workload.

*Presently with Kyoto University. (h.nakashima@media.kyoto-u.ac.jp)

This *worst case interruption delay* (WCID) is very difficult to analyze because many factors are involved to determine it. First it is obviously required to know the WCET of a set of preemptors which are invoked by the interruption. Second we have to analyze the worst case scheduling of the preemptors and the interrupted process to determine the preemptor set. Finally and most challengingly, we cannot assume the CPU time consumed by the interrupted process is as same as that without interrupt because caches and branch predictors are *polluted* by preemptors and, from a microscopic viewpoint, instruction pipeline is *flushed*.

Since modern microprocessors have complicated mechanisms of instruction scheduling, it is not sufficient to find the interruption point which maximizes the number of cache misses and/or branch prediction misses. That is, the point may not be worst because the delay caused by the misses may be hidden by out-of-order scheduling while the other point with a less number of misses is more harmful due to tightly dependent instructions executed after it.

The aim of our research is to find a tight and safe upper bound of the delay caused by one interruption by directly simulating a workload execution on a cycle accurate simulator (CAS). Our method is highly accurate because it is equivalent to a huge set of simulations with all possible cases of interruption points. The most important contribution of this paper is to give an efficient algorithm of $O(N(K + \log N))$ time complexity, where $N$ is the number of executed instructions, and $K$ is a large constant for out-of-order scheduling simulation and is usually dominant over the $O(\log N)$ factor for a simple binary tree traversal.

The rest of paper describes our WCID analyzer as follows. First the key idea of efficiency, *differential simulation* is introduced in Section 2 after modeling interruptions and target processors. Our implementation based on SimpleScalar[1] is described in Section 3 together with experimental results with SPEC CPU95 benchmarks shown in Section 4. After a brief discussion of related work in Section 5, we conclude this paper in Section 6.

## 2. DIFFERENTIAL SIMULATION

This section describes the key idea of our method, *differential simulation*, which is based on the observation that two executions with different interruption points have a set of durations in which they behave *coherently*. Before introducing this idea, a model of target processors with its

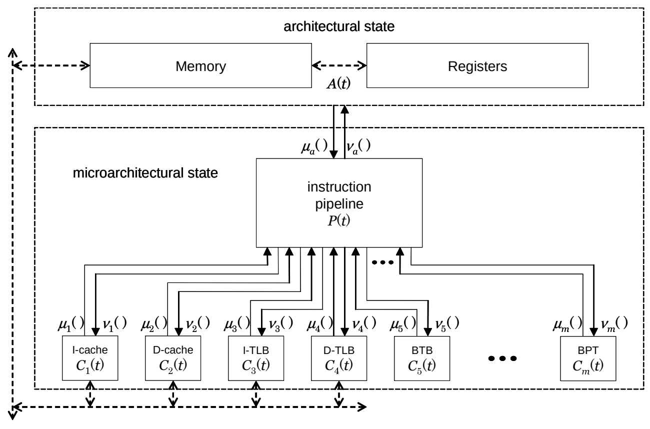


**Figure 1: Model of Target Machine**

conceptual behavior on an interrupt is given. Then, after showing a straightforward iterative simulation as the base of our idea, the differential simulation method is described.

## 2.1 Models of Processor and Interruption

Modern microprocessors with caches, branch predictors and an out-of-order instruction pipeline may be modeled as shown in Figure 1. A processor may be considered as a huge but finite state machine consisting of *architectural* and *microarchitectural* states.

The architectural state is usually represented by the combination of architectural registers and a (physical) memory. At any point in the execution of a workload, their contents are determined only by the sequence of instructions, $i_1, \ldots, i_n$ executed until the point and their initial values (e.g. 0 for all), and thus may be represented by a function of $n$, namely $A(n)$. Now let us assume the workload execution of $N$ total instructions is interrupted at $i_n$, i.e. just after the instruction $i_n$ completes, a set of preemptors is executed, and then the original execution is resumed for remaining $i_{n+1}, \ldots, i_N$. As far as the interrupted process concerns, the architectural state $A(n)$ remains unchanged during the execution of the preemptors[1]. Furthermore, for any pair of executions with different interruption points, namely $i_n$ and $i_{n'}$, their architectural states are considered equivalent to $A(k)$ for any $k$ independent from $n$ and $n'$.

On the other hand, the microarchitectural state at a machine cycle $t$, namely $M(t)$, is greatly affected by an interruption and its timing. Before discussing the effect, we decompose $M(t)$ into a series of (almost) independent states of *instruction pipeline* $P(t)$ and *cache-like modules* (CLM in short) $C_1(t), \ldots, C_m(t)$ for caches, TLBs, branch prediction tables, and so on. The pipeline state $P(t)$ represents instructions which has been fetched but has not retired (or been committed) yet, the pipeline stage where each instruction resides, the delay of stage progression of each instruction, and so on. The pipeline automaton also has output functions of its state $P$, $\nu_a(P)$ and $\nu_1(P), \ldots, \nu_m(P)$ to provide input symbols to architectural and CLM automata.

[1]A part of architectural state referred by preemptors should have been changed, of course, but this change will not affect the execution of interrupted process unless it interacts with preemptors. Although interactions among processes and operating system could be handled easily if necessary, we omit this issue in this paper.

The architectural automaton changes its state from $A(n(t))$ to $A(n(t+1))$ by $\nu_a(P(t))$ where $n(t)$ is the total number of *retired* (or committed) instructions until $t$, and outputs a symbol $b_a(t) = \mu_a(\nu_a(P(t)), A(n(t)))$ to provide instructions and data items to pipeline.

The state of a CLM $C_k(t)$ may be decomposed further into a series of independent substates $C_{k,1}(t), \ldots, C_{k,s}(t)$ corresponding to, for example, cache sets. Its input symbol $\nu_k(P(t))$ only affects (at most) only one substate, namely $C_{k,j}(t)$ and changes it by $C_{k,j}(t+1) = \lambda_k(\nu_k(P(t)), C_{k,j}(t))$. For example, $\nu_k(P(t))$ represents an address and access mode (e.g. read or write) to a cache, and changes the substate of the cache set for the address if it causes a replacement, recently-used reordering, and so on. A CLM also outputs a symbol $b_k(t) = \mu_k(\nu_k(P(t)), C_k(t))$ to provide the result of $\nu_k(P(t))$, e.g. hit or miss, delay of the access, to pipeline. Finally, the pipeline automaton changes its state by $P(t+1) = \lambda_p(b_a(t), b_1(t), \ldots, b_m(t), P(t))$.

Now we model the effect of an interruption at $i_n$ on each microarchitectural component. Since we try to estimate the *worst case* delay due to the interrupt, each state at the resumption of $i_{n+1}$ at cycle $t$ is defined as follows regardless of preemptors.

- Pipeline state $P(t)$ is *emptied* and thus has no instructions. Therefore, we have to resume the execution from the instruction fetch for $i_{n+1}$.
- Each CLM state $C_k(t)$ must be set to a value that maximizes the execution cycles for $i_{n+1}, \ldots, i_N$. A reasonable approximation for caches, TLBs and branch target buffers (BTBs) is to *invalidate* all the substates of them so that the first (and subsequent a few if set-associative) access causes a miss(-prediction). The worst case values of a branch direction predictor, which does not have such an apparent ones, will be discussed in Section3.3.

## 2.2 Straightforward Iterative Simulation

It is easy to estimate WCID in a straightforward manner in which a workload is simulated iteratively varying interruption point as shown in Figure 2. We define an instance of the workload execution of $i_1, \ldots, i_N$ interrupted at $i_n$ as a *thread* for $i_n$ and notate it as $T_n$. We also define $T_0$ as a special thread without any interruptions. With these definitions, the straightforward algorithm is outlined as follows.

1. Do a *cycle accurate simulation* (CAS) for $T_0$ logging the cycle count from the beginning, $\tau_j^0$, each time the instruction $i_j$ retires (topmost bold arrow in the figure). Note that $\tau_{j+1}^0 - 1$ is the worst case cycle count for the execution preceding the interruption at $i_j$.
2. For each thread $T_j$ $(1 \le j < N)$ do the following. First, obtain the architectural state $A(j)$ by an architectural *fast-forwarding* simulation (thin arrows in the figure). Then set the microarchitectural state $M^j(0)$ to that defined in Section 2.1 (star marks in the figure), and do CAS for $T_j$ from $i_{j+1}$ to $i_N$ to have its total cycle count $\tau_N^j$ (second and subsequent bold arrows in the figure).
3. The WCET with one interrupt, namely $\tau_w$, is given by;

$$\tau_w = \max_{0 \le j < N} \{\tau_{j+1}^0 - 1 + \tau_N^j\}$$

and WCID is given by $\tau_w - \tau_N^0$.

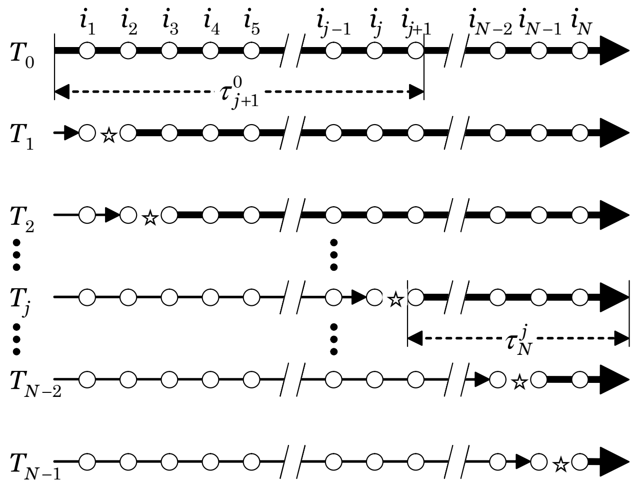


**Figure 2: Straightforward Iterative Simulation**

It is obvious that this straightforward algorithms takes $O(N^2)$ time. For example, even for a tiny workload of $N = 10^6$, a CAS of 1 MIPS takes $0.5 \times 10^6$ second or about six days to simulate $0.5 \times 10^{12}$ instructions. Note that it is hard to do the work with a real machine due to the difficulty of the *simulation* of interruption at each instruction execution. Moreover, even if this is possible with a trick using hardware counters and intentional pollution of CLMs, 1 GIPS machine at least takes 1,000 second for the work of $N = 10^6$, and 100,000 second or 28 hours if $N$ grows to $10^7$.

## 2.3 Outline of Differential Simulation

The basic idea of the differential simulation is that the executions of adjacent threads are expected to behave *similarly.* For example, Figure 3 illustrates abstracted behavior of two threads $T_{j-1}$ and $T_j$ where $i_j$ is a jump-and-link (`jal`) instruction residing at address $\alpha$ to call a function `F()` starting from address $\beta$. When the thread $T_{j-1}$ starts the fetch of the first instruction of `F()` (i.e. $i_{j+1}$) at its cycle $t_0^{j-1}$, its microarchitectural state $M^{j-1}(t_0^{j-1})$ has the following difference from $T_j$'s initial state $M^j(t_0^j)$ ($t_0^j = 0$) as follows[2].

(1) $P^{j-1}(t_0^{j-1})$ may have $i_j =$ `jal` while $P^j(t_0^j)$ does not have anything.

(2) Instruction cache, namely $C_{ic}^{j-1}(t_0^{j-1})$, has instructions spatially following to $i_j$, `xxx`, `yyy` and so on providing they reside in the memory block where $i_j$ does, but $C_{ic}^j(t_0^j)$ does not have any useful instructions.

(3) Branch target buffer, namely $C_{btb}^{j-1}(t_0^{j-1})$, has the address $\beta$ of the first instruction of `F()` as the target of $i_j =$ `jal`. Return address stack, namely, $C_{ras}^{j-1}(t_0^{j-1})$, has the address $\alpha+1$ of `xxx` as its top-of-stack element. On the other hand, neither $C_{btb}^j(t_0^j)$ nor $C_{ras}^j(t_0^j)$ have any useful target addresses.

[2]The explanation below omits the following microscopic behavior of $T_{j-1}$ for the sake of simplicity. (1) Since BTB has nothing when $i_j =$ `jal` is fetched, $T_{j-1}$ must mispredict its target and thus may fetch some instructions from the bogus target filling instruction cache with them. (2) Although BTB may not have had the address of the first instruction of `F()` yet when $T_{j-1}$ starts the fetch of it depending on the BTB update mechanism, the address will be registered into BTB soon.

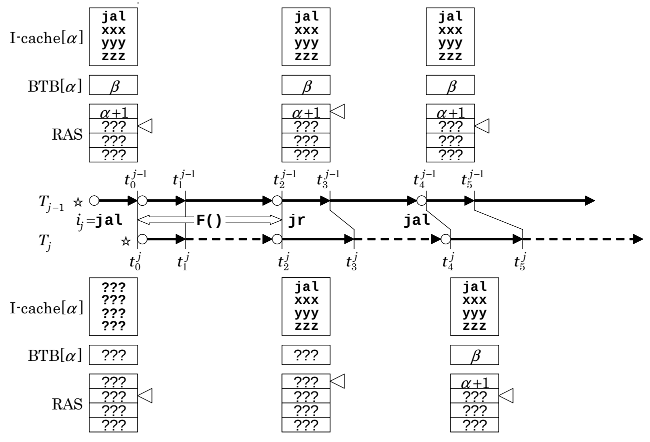


**Figure 3: Differential Simulation**

The differences above, however, will resolve on the way of the progress of both threads as follows.

(a) Since $P^{j-1}(t_0^j)$ only has $i_j =$ `jal`, the progress of `F()`'s instructions in the pipeline will not be affected by the existence of $i_j$. Therefore, when $i_j$ retires at $t_1^{j-1}$, $P^{j-1}(t_1^{j-1})$ will have `F()`'s instructions in a form as if $P^{j-1}(t_0^j)$ was empty and thus it is equivalent to $P^j(t_1^j)$ where $t_1^j = t_0^j + (t_1^{j-1} - t_0^{j-1})$.

Even if $P^{j-1}$ does not behave as this hypothesis expects, two pipelines become equivalent soon unless one of them *touches* a CLM substate different from the other because of the following. A pipeline becomes *sparse* when it has instructions with long latency due to cache and branch prediction misses. Since the progress of instructions in a sparse pipeline is hardly affected by preceding instructions in it, the state of the pipeline tends to be determined by instructions currently residing rather than its old *memory.* Moreover, the pipeline becomes almost empty when a branch prediction miss occurs, and thus almost completely *forgets* its old memory. Therefore, it is strongly expected that we have some $t_{1'}^{j-1}$ and $t_{1'}^j$, such that $P^{j-1}(t_{1'}^{j-1}) = P^j(t_{1'}^j)$ if `F()` has a sufficiently large number of instructions, since the CLM differences shown in (2) and (3) are not touched by both threads.

(b) After two pipelines become equivalent at $t_1^{j-1}$ and $t_1^j$, they are kept equivalent until the last instruction of `F()` is fetched at $t_2^{j-1}$ and $t_2^j$. That is, $P^{j-1}(t^{j-1}) = P^j(t^j)$ holds for all $t^{j-1}$ and $t^j$ such that $t_1^{j-1} \le t^{j-1} \le t_2^{j-1}$ and $t_1^j \le t^j \le t_2^j$, because CLM differences shown in (2) and (3) are not touched by both threads and thus $\nu_x(P^{j-1}(t^{j-1})) = \nu_x(P^j(t^j))$ and $b_x^{j-1}(t^{j-1}) = b_x^j(t^j)$ for all $x \in \{a, 1, \dots, s\}$. We refer this duration as to that $T_j$ is *coherently* executed with $T_{j-1}$. The dashed arrow in the figure represents this coherent execution of $T_j$ whose simulation may be omitted because we know $P^j(t_2^j) = P^j(t_2^{j-1})$ and $t_2^j - t_1^j = t_2^{j-1} - t_1^{j-1}$.

(c) The coherence is broken by the fetch of `F()`'s last instruction jump-on-register (`jr`) to return from `F()`.

Firstly $T_{j-1}$ correctly predicts the return address by popping $C^{j-1}_{ras}$ unless `F()` has too deep function nesting, but the pop of $C^{j}_{ras}$ simply gives us an invalid prediction result. Secondly the instructions spatially following $i_j$ = `jal` resides in $C^{j-1}_{ic}$ unless they are replaced in `F()`, while accessing $C^{j}_{ic}$ for them causes a miss after $T_j$ recognizes the misprediction. However, these operations makes $C^{j}_{ic}$ equivalent to $C^{j-1}_{ic}$ because they both have the instructions in problem now[3]. Then, as discussed in (a), both pipelines become equivalent again at some cycles $t^{j-1}_3$ and $t^j_3$.

(d) $T_j$ is coherently executed with $T_{j-1}$ again, until it encounters a `jal` at $\alpha$ for `F()` again, or until the end of workload execution if $i_j$ is for the last call of `F()`.

(e) The coherence is broken again at the second fetch of `jal` at $t^{j-1}_4$ and $t^j_4$, because $T_{j-1}$ correctly predicts its target while $T_j$ cannot. This misprediction, however, makes $C^{j-1}_{btb}$ equivalent to $C^{j}_{btb}$. Moreover, the execution of `jal` makes $C^{j}_{ras}$ equivalent to $C^{j-1}_{ras}$ because they both have the address following the second `jal`. Then, both pipelines become equivalent once again at some cycles $t^{j-1}_5$ and $t^j_5$.

(f) Now we have $M^{j-1}(t^{j-1}_5) = M^j(t^j_5)$ that means $T_j$ may no longer be simulated because its execution is apparently coherent with $T_{j-1}$ hereafter.

Our idea of *differential simulation* based on the scenario above is to simulate $T_j$'s behavior only for the durations *different* from its predecessor $T_{j-1}$ (solid arrows in the figure) omitting those coherent with $T_{j-1}$ (dashed arrows in the figure). An important observation obtained from the scenario is that the amount of the initial difference between $M^{j-1}$ and $M^j$ is bounded to some *constant*. More importantly, it is strongly expected that the number and total length of the durations to be simulated differentially are also bounded to some *constant* independent from $N$.

If these observations are correct, the thread $T_0$ without interruption solely requires out-of-order simulation of $N$ instructions, while other $N-1$ threads needs that of $K$ instructions at most where $K$ is some constant determined by microarchitecture and workload characteristic but independent from $N$. Therefore, as far as the simulated instruction count concerns, WCID can be calculated in $O(N)$ time if we can construct the architectural state at $i_j$ for each $j$ in a constant time as described later.

## 3. IMPLEMENTATION

This section describes our method to implement the idea shown in Section 2.3 using SimpleScalar-3.0[1] as the base. First, Section 3.1 shows how each thread is managed for the differential simulation. Then, the mechanism to manage the differences of CLM substates for each thread is explained in Section 3.2. Next, special treatment for 2-bit counter based branch predictors is discussed in Section 3.3. Finally, Section 3.4 explains an algorithm to count cycles of *sleeping* threads whose executions are coherent with others using a binary tree, which is the sole non-linear portion of our WCID analysis.

[3]$C^j_{ic}$ could have instructions on the mispredicted execution path but we ignore them for the sake of explanation simplicity.

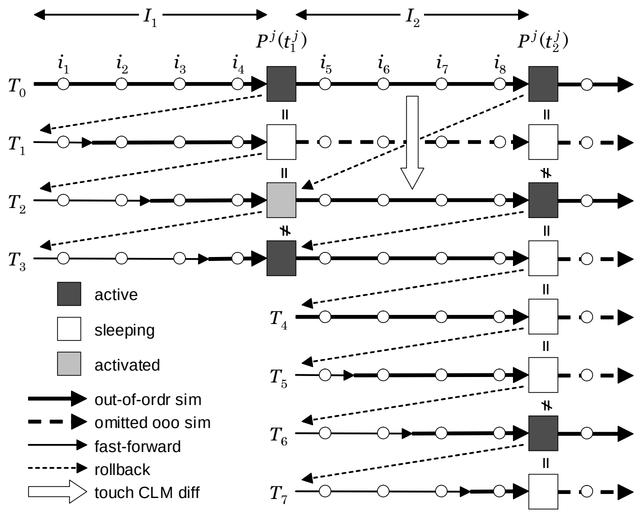


**Figure 4: Thread Simulation in Interval**

### 3.1 Thread Management

In order to minimize the length of durations of out-of-order simulation for a thread, its pipeline state must be compared with its predecessor as frequently as possible. On the other hand, too frequent comparison should incur a large overhead and degrade the total performance. Therefore, we compare the states periodically with a predefined small *interval* $N_I$ of executed instruction counts, rather than at every instruction or cycle.

Before explaining the thread management with intervals, we define how we count *executed* instructions more specifically. SimpleScalar's out-of-order simulator `sim-outorder` is categorized into *functional-first* type ones in which the architectural state is updated when an instruction is decoded (or dispatched) instead of its retirement. Since we need to compare pipeline states of two threads when their architectural states are equivalent, we count instruction executions at decoding to detect the end of an interval. Furthermore, since the functional-first technique makes it possible to examine the correctness of branch/jump target prediction at decoding, we count instructions on *true paths* so that the pipeline back-end does not have any instructions on *false paths* at the end of an interval.

Figure 4 shows the first two intervals, $I_1$ and $I_2$, of four instructions rather than eight in real implementation. In the first interval $I_1$, four threads, $T_0$ to $T_3$, are *created* and simulated in ascending order. That is, we first simulate $T_0$ and *suspend* it just after four instructions have passed the decoding stage[4], saving its clock cycle $t^0_1$, the contents of architectural registers and pipeline state $P^0(t^0_1)$ for the next interval.

Then, we *rollback* the architectural state to the beginning of $I_1$ by setting architectural registers to their initial values. As for the rollback of memory state, we have a stack into

[4]Since multiple instructions, up to four in our experiment, may be decoded in one cycle, the real suspension condition is that four or *more* instructions have passed the stage. Thus when we resume an *overruning* thread, a short instruction-level simulation to regain its architectural state is preceded. This microscopic issue is omitted in the following explanation for the sake of simplicity.

**Table 1: Pipeline Components of SimpleScalar**

| queues | depth |
|---|---|
| instruction fetch queue | 4 |
| register update unit | 16 |
| load/store queue | 8 |
| ILP | width |
| instruction fetch | 4 |
| decode | 4 |
| commit | 4 |
| func. units | parameters (*1) |
| int ALU | $4 \times (1/1)$ |
| fp ALU | $4 \times (1/2)$ |
| int mult/div | $1 \times (\text{mult} = 1/3;\ \text{div} = 19/20)$ |
| fp mult/div | $1 \times (\text{mult} = 1/4;\ \text{div} = 12/12)$ |
| memory ports | $2 \times (1/c)$ |

(*1) $n \times (t/l)$: $n$ = number of units; $t$ = throughput; $l$ = latency

Memory port latency $c$ is determined by cache.

which each store operation executed in $T_0$ pushes its address and the value before update. Thus on the suspension of $T_0$, the stack is traversed from its top to bottom restoring saved values to regain the memory state at the beginning of $I_1$.

Before the simulation of $T_1$, we perform an instruction-level *fast-forwarding* simulation of the first instruction $i_1$ to have the architectural state at its completion. Then the out-of-order simulation takes place and $T_1$ is suspended at its cycle $t_1^1$. On the suspension, $T_1$'s pipeline $P^1(t_1^1)$ is compared with $P^0(t_1^0)$. More specifically, we compare the contents of SimpleScalar's components shown in Table 1 in which their default configuration parameters are also shown. The comparison is fairly simple but we have to pay some attention to *absolute* timing values in the components. That is, a pair of timings at which some event will occur, e.g. the completion of an instruction, has to be compared after converting each of them into the value relative to own cycle count of each thread, $t_1^0$ for $T_0$ and $t_1^1$ for $T_1$.

If the comparison results that both pipelines are equivalent, the thread $T_1$ *sleeps* until a CLM substate is touched to break the coherency with $T_0$ as discussed later. A sleeping thread is called being *dominated* by the thread coherent with it. In the figure, $T_1$ is dominated by $T_0$. Then the simulation and suspension are repeated for $T_2$ and $T_3$. In the figure, $T_2$ also sleeps and thus is dominated by $T_0$. On the other hand, $T_3$ is kept *active* because its pipeline $P^3(t_1^3)$ has some difference from $P^0(t_1^0)$.

Now we finished the interval $I_1$ in which $N_I(N_I+1)/2$ instructions are simulated in out-of-order manner while $N_I(N_I - 1)/2$ are fast-forwarded. Note that these amounts and number of operations for rollback are *constant*. Then we proceed to the second interval $I_2$ in which new four threads, $T_4$ to $T_7$, are created. Before simulating them, we resume the simulations of active threads $T_0$ and $T_3$. Since $T_0$ dominates $T_1$ and $T_2$, we have to take care the possibility that $T_0$ touches a CLM substate different from those of $T_1$ and $T_2$. The method to check this coherence breaking is discussed in Section 3.2. The figure shows the case in which $T_2$ is *activated* by the difference and its simulation is resumed from the beginning of $I_2$. For this resumption, the pipeline state $P^2(t_1^2)$ is replicated from $P^0(t_1^0)$ and timing values in it is *adjusted* by $t_1^2$ (see footnote[5])

Then we simulate $T_3$, $T_4$, ..., $T_7$ and suspend them to finish the interval $I_2$. In the figure, $T_0$, $T_2$ and $T_6$ are active

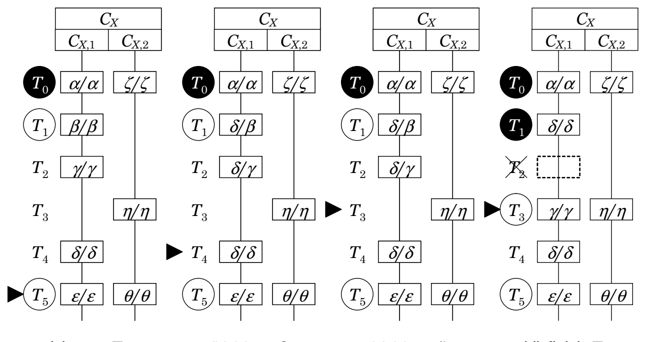


**Figure 5: Accesses to CLM substates**

at the end of $I_2$ while others are sleeping and are dominated by active ones. ).

## 3.2 CLM substate management

Each thread may have its own CLM substates different from those of its predecessors. When a thread $T_j$ is sleeping, its dominator $T_k$ $(k < j)$ is responsible to examine if an access to a CLM substate $C_{l,m}^j$ for $T_j$ gives a result different from that of $C_{l,m}^k$ for $T_k$ to break their coherency and thus to activate $T_j$.

For this examination, each CLM substate is represented in a linked list of nodes each of which corresponds to the substate value of a thread's own. For example, Figure 5(a) shows a small CLM named $C_X$ of two substates $C_{X,1}$ and $C_{X,2}$ accessed in an interval $I$ in this order. Threads $T_0$ to $T_5$ may have their own substates whose values are represented as $x/y$ where $x$ is the *current* value while $y$ is the value at the end of the last interval that threads passed. Threads except for $T_3$ have their own substate of $C_{X,1}$, and threads $T_0$, $T_3$ and $T_5$ have those of $C_{X,2}$. Thus for $T_3$, the value of $C_{X,1}$ is $\gamma/\gamma$ for its predecessor $T_2$. The thread $T_0$ has finished $I$ and is active (represented by black circles), threads $T_1$ and $T_5$ are also active but have not yet started $I$ (represented by white circle), and other threads are sleeping and dominated by $T_1$.

Now we start the simulation for $T_1$ in the interval $I$. First it performs an access to $C_{X,1}$ which *hits* if the substate value is $\delta$. First $T_1$ examines its own substate and finds it results miss. Thus the *current* value of the substate is changed to $\delta$ but its *old* value remains unchanged. Then $T_1$ traverses all substate values, $\gamma/\gamma$ and $\delta/\delta$, for threads dominated by it. Although the value $\gamma/\gamma$ for $T_2$ (and $T_3$) is different from $\beta/\beta$ for $T_1$, the access results affected to pipeline, $b_x^1 = \mu_x(\nu_x(P^1), C_x^1)$ and $b_x^2 = \mu_x(\nu_x(P^1), C_x^2)$, may be equivalent as we assume in this explanation. Thus the substate for $T_2$ is changed to $\delta/\gamma$ but $T_2$ (and $T_3$) is not activated. The substate $\delta/\delta$ for $T_4$, however, has a different story because the hitting access to it should give a result different from the missing access to $\beta/\beta$. Therefore, we now find that the coherence of $T_1$ and $T_4$ is broken, and $T_4$ becomes the *candidate* of activation as indicated by a tri-

[5]In this scenario, replicating of $P^0(t_1^0)$ for $P^2(t_1^2)$ is unnecessary because $T_2$ has just slept and thus $P^2(t_1^2)$ is valid. In general, however, since an activated thread may have slept during many intervals, the pipeline at its bedtime should be obsolete on its uprising.

**Table 2: CLM of SimpleScalar**

| caches / TLBs | parameters (*1) |
|---|---|
| L1 cache | separated / unified / none |
| | $\underline{i = 32 \times 1 \times 512, \quad d = 32 \times 4 \times 128}$ |
| L2 cache | separated / unified / none |
| | $\underline{64 \times 4 \times 1024}$ |
| TLB | separated / unified / none |
| | $\underline{i = 4096 \times 4 \times 16, \quad d = 4096 \times 4 \times 32}$ |
| predictors | parameters (*2) |
| BTB | $\underline{4 \times 512}$ |
| ret addr stack | $\underline{8 \times 1}$ |
| dir predictor | not-taken / taken / perfect / |
| | bimodal / gselect / gshare / combined |
| | $\underline{2 \times 2048}$ |

(*1) $b \times w \times s$: $b$ = block / page size in byte; $w$ = associativity; $s$ = # of sets (substates)
(*2) $x \times s$: $x$ = associativity / stack depth / counter width; $s$ = # of entries (substates)

angle shown in Figure 5(b) which also shows the changes of substates.

Then $T_1$ accesses $C_{X,2}$ referring to its predecessor's substate $\zeta/\zeta$. Since the access hits to $\zeta$, the substate remains unchanged[6]. Then it examines $\eta/\eta$ for $T_3$ to find that coherence is broken again. Thus $T_3$ becomes the new activation candidate as shown in Figure 5(c).

Finally, when we finish $T_1$'s simulation of the interval $I$, each accessed substate is traversed again. First, those of $C_{X,1}$ are traversed to perform the following; $T_1$'s substate $\delta/\beta$ is changed to $\delta/\delta$ because $\delta$ is now *old* value for the next interval $I+1$; $T_2$'s substate $\delta/\gamma$ is removed because it is now equivalent to $T_1$'s; $T_2$ itself is also removed because its microarchitectural state including CLM state is now equivalent to $T_1$'s; $T_3$'s substate $\gamma/\gamma$ is newly created and inserted copying $T_2$'s old value $\gamma$, because $T_3$ must start from the beginning of $I$; and finally $\delta/\delta$ for $T_4$ remains unchanged because it has not been modified[7]. Then we traverse substate values of $C_{X,2}$ but no operations are performed because they have not been modified. Now $T_3$ is ready to be simulated from the beginning of $I$ with its own substate values $\gamma/\gamma$ and $\eta/\eta$.

As explained above, a thread simulation in an interval needs to traverse substate values for all threads dominated by the thread. Although the number of sleeping threads is hardly bounded to a constant, it is strongly expected that the number of traversed substate values are bounded to a constant in practical workloads. For example, let us assume a substate of 2-way set-associative cache is accessed with addresses $\alpha$ and $\beta$ in this order and their tag parts are different from each other. For all threads which started before this access sequence, the substate should have $\beta \to \alpha$ where $\to$ represents recently-used precedence. For other threads invoked after the access with $\alpha$, the value will be $\beta \to \perp$ or $\perp \to \perp$ where $\perp$ means an invalid block. Therefore, by the substate comparison and removal at the end of each interval, the number of substate values should be kept to three or less effectively.

[6]If the access makes the substate changed to, say $\kappa$, a new node $\kappa/\zeta$ would be *inserted* just below $\zeta/\zeta$ for $T_0$ so that $T_1$ has its own substate.

[7]If it has been modified, we restore its old value into the current value to make it $\delta/\delta$.

Table 2 shows all kinds of CLM which SimpleScalar supports and thus we implemented as well. Underlined configurations are SimpleScalar's defaults and are chosen for our experiment discussed in Section 4.

## 3.3 Worst Case Values of 2-Bit Counter Based Predictors

Caches, TLBs, BTBs and return address stacks have apparent worst case values to which their substates should be set on interrupt. That is, all of their substates may be set to an *invalid* value[8] to maximize the number of misses in the execution following an interrupt. Branch direction predictors with 2-bit counter tables, however, do not have such an apparent value because each of four possible values has some significance for the branches referring to it. More complicatedly, the value to cause prediction miss at the first branch after an interruption may not be worst. For example, a sequence of branches NTT(NT)$^*$, where N and T mean not-taken and taken, is suffered the maximum number of misses when the counter referred by them has 1 at initial but this value predicts the direction of the first non-taken branch correctly.

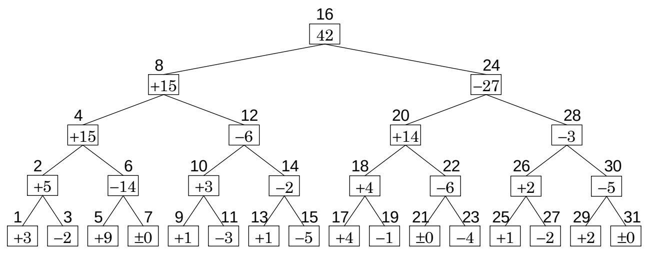
Thus we have to determine the worst case values of counters for each interruption point so that they maximize number of mispredictions. Since determining them with perfect accuracy needs a huge amount of exhaustive search with out-of-order simulation in $4 \times B$ search space for prediction table of $B$ entries, we adopt an approximation obtained by an analysis of *in-order* instruction level simulation result. That is, in advance of WCID analysis, we perform a fast instruction level simulation to obtain a trace of conditional branches $b_1, \ldots, b_n$. Then for each branch $b_k$, we analytically obtain the worst case value of the counter it refers to as follows.

Let $c(k)$ be the counter which $b_k$ refers to, $p(k)$ be the index of the branch which also refers to $c(k)$ and just preceding $b_k$, and $m_c^\gamma(k)$ be the number of misses which branches in the set $\{b_l \mid k < l \le n,\ c(l) = c\}$ suffer if $c$ has the value $\gamma$ $(0 \le \gamma \le 3)$ at the completion of $b_k$. From these definitions, we have the following recurrence for $m_{c(k)}^0(k')$, for example, for all $k'$ such that $p(k) \le k' < k$ providing we know $m_{c(k)}^0(k)$ and $m_{c(k)}^1(k)$.

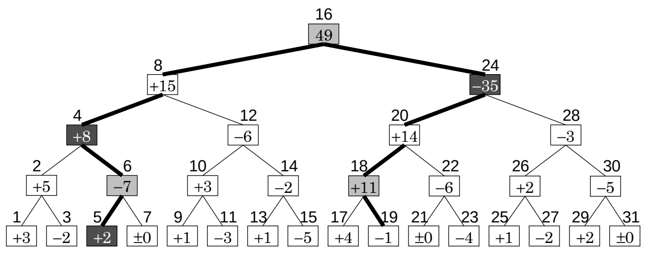
$$m_{c(k)}^0(k') = \begin{cases} m_{c(k)}^1(k) + 1 & \text{if } b_k \text{ is a taken branch} \\ m_{c(k)}^0(k) & \text{if } b_k \text{ is a not-taken branch} \end{cases}$$

That is, if $c(k)$ is set to 0 somewhere between $b_{p(k)}$ and $b_k$, and $b_k$ is a taken branch, $b_k$ is mispredicted and $c(k)$ becomes 1 to result that the total number of misprediction is $m_{c(k)}^1(k) + 1$. If $b_k$ is a not-taken one, its action is correctly predicted and $c(k)$ remains unchanged. Since similar recurrences are obtained for $\gamma \ge 1$, and $m_c^\gamma(n) = 0$ for any $c$ and $\gamma$ by definition, it is easy to calculate $m_c^\gamma(k)$ for all $c$, $\gamma$ and $k$ $(0 \le k \le n)$ providing $p(k) = 0$ for the first branch referring to $c(k)$. Thus, it is also easy to find $w(k)$ s.t. $m_{c(k)}^{w(k)}(k) = \max_\gamma\{m_{c(k)}^\gamma(k)\}$ which must be set to $c(k)$ on an interruption that occurs after $b_k$ and before $b_l$ such that $k = p(l)$. Note that this analysis simply requires $O(N)$ fast instruction level simulation and also $O(N)$ branch trace

[8]BTB and return address stack cannot be invalidated usually, but can have some key and/or target address which must cause prediction miss. For example, an address outside workload's text area may be used for this purpose.



(a) before addition



(b) after addition

**Figure 6: Maintenance of Thread Cycles using Binary Tree**

analysis. In our implementation, values of $w(k)$ are stored in a file which our WCID analyzer reads to set a CLM substate for a *bimodal* predictor[12] for the thread $T_j$ such that $i_j = b_k$.

Predictors using *global history* to generate 2-bit counter table index, namely *gselect*[10] and *gshare*[6], make the story a little bit complicated because we have to find not only worst case values but also worst case indices. That is, an interruption makes the global history of $g$ bits uncertain and thus the counter table indices for the following $g$ branches uncertain. Although a perfect analysis is possible by a search of a huge space, we adopt a more efficient method which results a little bit more mispredictions than the exhaustive search. For a predictor with $g$-bit history, we assume $g$ branches after an interruption always miss to predict their directions and make no change to the counter table. The table is set to the worst case pattern obtained from an analysis similar to that for bimodal predictors, *after* $g$ branches complete using the the history *fixed* by the branches. This method could be a little bit too pessimistic because the table pattern might be infeasible due to ignoring updates of leading $g$ branches, but this small pessimism should be compensated by the efficiency of the analysis.

Finally, a *combined* predictor[6], which has a *meta* predictor to choose the predictions from a bimodal table and a gselect or gshare table, is implemented with a fairly simple analysis based on the methods described above. Although it is required to calculate $2^{2+2+2} = 64$ possible values of $m_c^\gamma(k)$ for three counters referred by a branch, the time complexity of the analysis is still $O(N)$ and the constant factor is acceptably small.

## 3.4 Maintenance of Cycle Counts of Sleeping Threads

We have to not only simulate threads but also count cycles of thread executions to find WCID. Counting cycles of an active thread is of course obvious, but doing it for sleeping threads has a problem. The counting operation itself is fairly easy because we know the cycle when a thread went to sleep and the cycles spent by its dominator. However, since the number of sleeping threads might not be bounded by a constant but could be proportional to $N$, it could make the time complexity of our analysis $O(N^2)$ if we simply add the cycles spent by a dominator to the cycles of threads dominated by it each time the dominator finishes its simulation for an interval.

Thus we devised an $O(\log N)$ algorithm for the cycle maintenance of sleeping threads using a simple binary tree. As shown in Figure 6(a), cycle counts of threads, except for $T_0$, are kept in a binary tree whose nodes correspond to threads in a *differential* manner. Let $n_j$ be the node for the thread $T_j$ whose cycle count is denoted as $t^j$, and $p(j)$ be the thread index of the parent node of $n_j$. For example, $p(5)$ is 6, $p(6)$ is 4, and so on. The root node, $n_{16}$ in the figure, has the absolute value of the cycle $t^{16}$ for the corresponding thread $T_{16}$. A non-root node $n_j$, however, has the difference of cycles between its own and its parent's, namely $t^j - t^{p(j)}$. Thus the absolute cycle of $T_j$ is calculated by adding the values of its own and its ancestors up to the root paying $O(\log N)$ cost. For example, $t^6$ is obtained by adding the values of $n_6$, $n_4$, $n_8$ and $n_{16}$ to result $-14 + 15 + 15 + 42 = 58$.

Now suppose $T_6$ dominates threads $T_7$ to $T_{19}$ and it finishes the simulation of an interval spending 7 cycles. Thus $\delta = 7$ must be added to $t^6$, $t^7$, …, $t^{19}$. This addition is performed by the following operations to the nodes on the upward paths, bold branches in the Figure 6(b), from the left child of $n_6$, namely $n_5$, and $n_{19}$ that has no children, up to their *common ancestor*, namely $n_{16}$. In the following explanation, we denote the series of nodes corresponding to the threads in problem as $S$, being $\langle n_6, n_7, \ldots, n_{19}\rangle$ in our example. We also assume that the root has a virtual parent (super-root), $n_{32}$ in our example.

(1) If $n_j \in S$ but $n_{p(j)} \notin S$, increment $n_j$'s value by $\delta$ to make it $(t^j + \delta) - t^{p(j)}$. In our example, $n_6$, $n_{16}$ (with super-root assumption) and $n_{18}$ satisfiy this condition and thus $\delta = 7$ is added to their values.

(2) If $n_j \notin S$ but $n_{p(j)} \in S$, decrement $n_j$'s value by $\delta$ to make it $t^j - (t^{p(j)} + \delta)$. In our example, $n_4$, $n_5$ and $n_{24}$ satisfy this condition and thus $\delta = 7$ is subtracted from their values.

(3) Otherwise, i.e. if $n_j \in S \leftrightarrow n_{p(j)} \in S$ holds, keep $n_j$'s value unchanged because the difference between the values of $n_j$ and $n_{p(j)}$ is not affected by the increment. In our example, $n_8$ $(n_8, n_{16} \in S)$, $n_{19}$ $(n_{19}, n_{18} \in S)$ and $n_{20}$ $(n_{20}, n_{24} \notin S)$ satisfy this condtion.

The correctness of the algorithm shown above, i.e. the assurance that it keeps the invaliant that each node $n_j$ has $t^j - t^{p(j)}$, is proved by a few deductions from the property of binary trees showing that $n_k \in S \leftrightarrow n_{p(k)}$ holds for all $n_k$ excluded from the upward path to the common ancestor (see [8] for a formal proof). It is clear that the algorithm takes $O(\log N)$ time. Since we maintain the cycle tree at each end of a thread simulation for an interval, the total cost of the maintanance is $O(N \log N)$ providing that the number of active threads at an interval is bounded by a constant. The caluculation of the absolute cycle count is performed

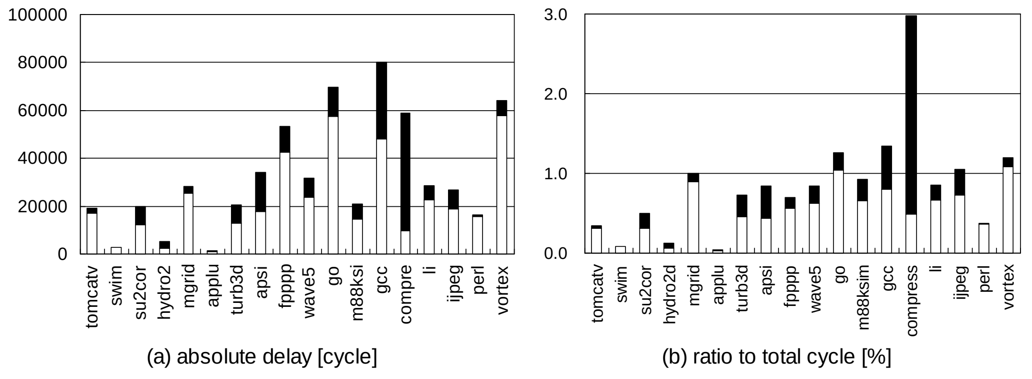


**Figure 7: Average and Worst Case Interruption Delays**

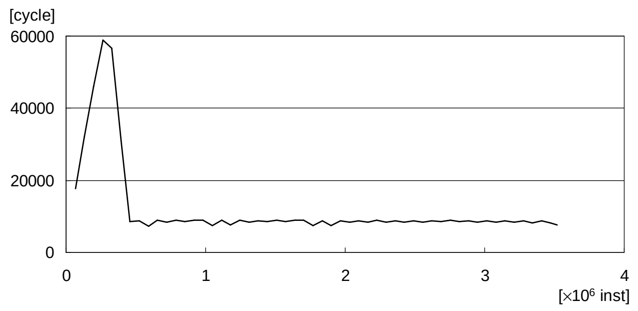


**Figure 8: Maximum Delay Caused by Interruptions in Each 64 K Instruction Duration of "compress"**

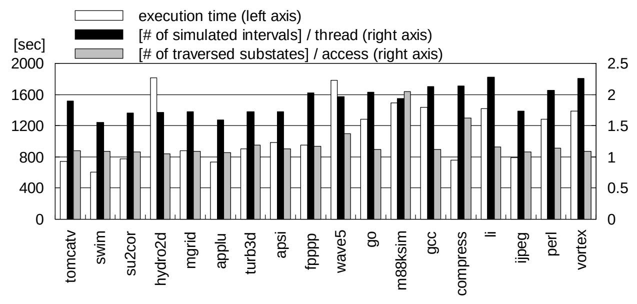


**Figure 9: Overall Execution Time, Simulated Intervals and Traversed CLM Substates**

when a thread is activated paying $O(\log N)$ cost, and thus the total cost for the calculation is $O(N \log N)$ again with the same hypothesis. Thus with the cycle maintenance algorithm described above, the time complexity of our WCID analysis is expected to be $O(N \log N)$ if our differential simulation successfully bounds the number of active threads in an interval to a constant.

Note that *growing* the tree upward adding new root and its right subtree is easy because the threads corresponding to added nodes are newly created and thus their absolute cycle counts are zero. For example, when $T_{32}$ is newly created, we simply add $n_{32}$ as the new root and $n_{33}$ to $n_{63}$ as its right subtree setting node values to zero. Since $t^{16} - t^{p(16)} = t^{16} - t^{32} = t^{16} - 0 = t^{16}$, the value of $n_{16}$ may remain unchanged.

## 4. EXPERIMENT

### 4.1 Environment and Workload

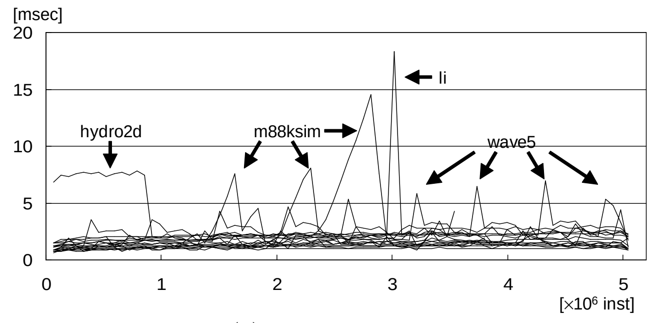
We implemented our WCID analyzer using SimpleScalar-3.0 as its base. The program is written in C as SimpleScalar is and is compiled by gcc 2.95.3 to run on an x86-based PC with Linux kernel 2.4.22. The performance is mesured using a 3 GHz Pentium-4 based PC with 1 GB memory. The target microarchitecture is SimpleScalar's default whose configuration parameters were shown in Table 1 and 2 in the previous section.

The workloads are all 16 programs in SPEC CPU95, compiled targeting SimpleScalar's PISA instruction set, with "test" data set. The workloads, except for ijpeg, are not for realtime systems but their wide range of behavioral spectrum will be helpful to examine the applicabilty of our analysis method. Although our analysis is expected to be reasonablly efficient, completely executing them would take too long time. In fact it would be longer than three weeks because the analysis speed is at most 8 times as slow as SimpleScalars `sim-outorder` with our setting of the interval length $N_I$ to 8, and `sim-outorder` takes about 2.5 days on our experimental environment. Thus we extracted five million instructions of the *midst* durations of their executions, except for two small workloads, m88ksim and compress, whose 3.8 and 3.5 million instructions are completely simulated.

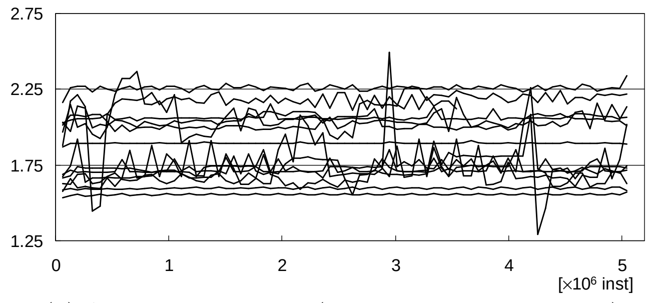
### 4.2 WCID Analysis Result

As the first result, WCID of each program is shown in Figure 7 in (a) absolute cycles and (b) ratio to the whole execution cycles. These graphs also show average interruption delays by white portions of bars. As easily expected, absolute and relative WCID vary in a wide range, from 1,411 (applu) to 80,292 cycle (gcc) and from 0.03 % (applu) to 2.98 % (compress), reflecting the characteristics of workloads.

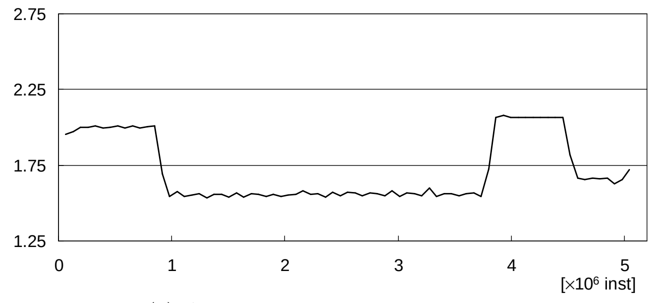
Although it is hard to evaluate whether the WCID values are significant or not, the remarkable result of compress exhibits the importance of detailed anaylsis. That is, its average delay is at the same level as other programs while its worst-case delay is significantly larger than others in terms of the ratio to its whole cycles. Thus, if we estimated the worst-case value from its medial and/or representative behavior, we could heavily underestimate the value resulting



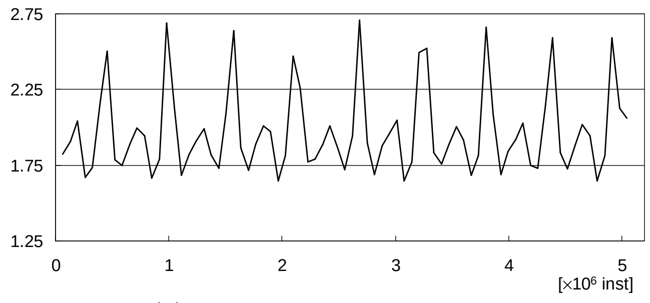
(a) Execution Time



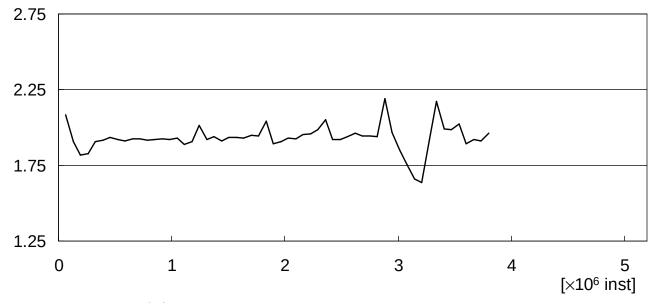
(b) Active Thread Ratio (Excluding Four Workloads)



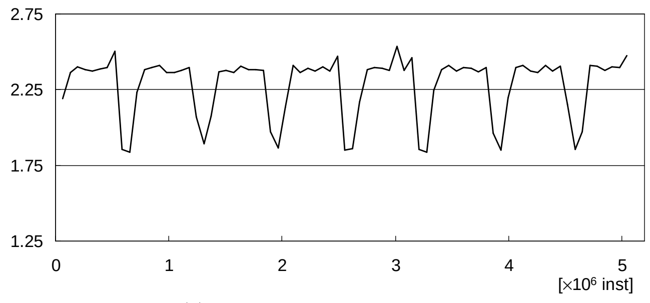
(c) Active Thread Ratio of hydro2d



(d) Active Thread Ratio of wave5



(e) Active Thread Ratio of m88ksim



(f) Active Thread Ratio of li

**Figure 10: Execution Time and Active Thread Ratio in Each Interval**

too optimistic estimation. In fact, as shown in Figure 8 in which the maximum delay caused by interruptions in each duration of 65,536 instructions are illustrated, its interruption delay strongly depends on the interruption timing and thus it should be difficult to estimate the worst-case dealy by a *run-through* investigation.

## 4.3 Simulation Time and Related Peformance Numbers

Since the most important expected feature of our WCID analyzer is its efficiency achieved by the differential simulation, measuring the performance numbers of its execution time is essential. First we show basic overall numbers in Figure 9 in which wall-clock execution times (white bars) are illustrated together with two important indicators which determine efficiency; average number of intervals in which a thread is simulated actively (black bars); and average number of CLM substates which are traversed in an access (gray bars).

The numbers are quite satisfactory. First, the execution times ranged from 608 second (swim) to 1814 second (hydro2d) are short enough for practical use which cannot be achieved by the straightforward $O(N^2)$ method. For example, sim-outorder which runs on our environment with 0.5 to 0.7 MIPS should take up to about 300 *days* to analyze one workload executing $12.5 \times 10^{12}$ instructions. Even if we did the analysis with a real machine of 1 GIPS, it would take 25,000 second or about 7 hours for $25 \times 10^{12}$ instructions.

The number of intervals, varying in the range from 1.55 (swim) to 2.28 (li), is also good because it means each thread only executes about 16 instructions on average, 8 in the interval of its creation and another 8 in some other interval mainly because of the activation by its dominator. The number of CLM substate traversal varies in the range from 1.05 (hydro2d) to 2.14 (compress). This means that almost only one additional access is required at most to a dominator to check the coherency of its dominating threads.

Althogh the overall numbers are good, this result does not assure that our algorithm works $O(N)$ simulation time and $O(N \log N)$ cycle maintenance time. Thus we measured three performance numbers discussed above for each of up to 625,000 intervals to observe *differential* behavior. The results are shown in Figure 10(a) to (f) and Figure 11. The execution times shown in (a) almost stably lie in the range from 1 ms to 3 ms but occasionally raise up to form the peaks of m88ksim, li and wave5 and a plateau of hydro2d. These

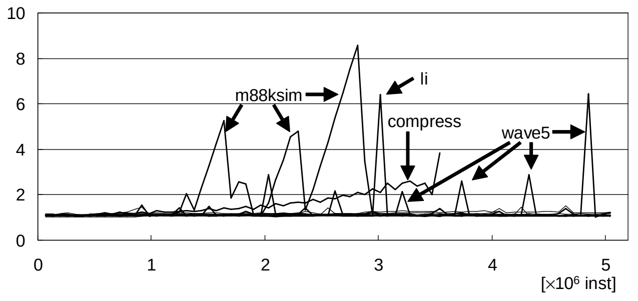


**Figure 11: Traversed CLM Substates in Each Interval**

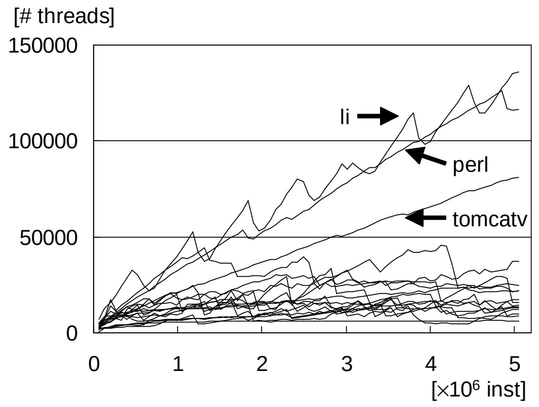


**Figure 12: Number of Sleeping Threads in Each Interval**

peaks and plateau, however, look reflecting the compuation phase shifts, because these unusual slowdowns disappear after one million instructions are executed.

This hypothesis that a program phase shift makes our analyzer slower only transiently is supported by the ratio of the number of active threads to created shown in (b) to (f), and substate traversals shown in Figure 11. Figure 10(b) illustrates the active thread ratio of workloads other than four which show peaks/plateaus, from which fairly stable behaviors are observed. As for hydro2d shown in (c), two plateaus are observed and the first one reflects that of execution time in (a). However, although other three shown in (d) to (f) exhibit periodical phase shifts, they hardly explain the peaks of execution time in (a). On the other hand, the number of substate traversals shown in Figure 11 matches and thus clearly explains the peaks in Figure 10(a). A possibly bad news that Figure 11 tells us is that the number in compress looks steadily growing, but this may simply show the first portion of a long but trasient phase shift.

Finally, we measured the number of sleeping threads in each interval, which is shown in Figure 12. The figure clearly shows that the number is propotional to $N$ at least in three workloads li, perl and tomcatv. This means that the analysis would take $O(N^2)$ time if we had executed threads whose machine states disagree with others or had maintained the cycle counts of sleeping threads in a straightforward manner propotional to the number of them.

From those results shown in Figure 10 and 12, we may conclude that our analyzer efficiently works in $O(N(K+\log N))$ time where $K$ is a significantly large constant hiding the effect of $O(\log N)$ factor, owing to our differential simulation and cycle count maintenance using binary tree.

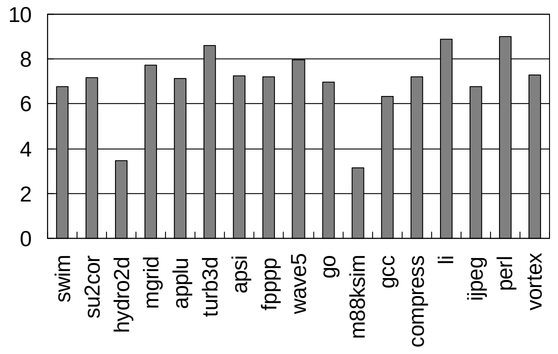


**Figure 13: Speedup of 8-node Parallel Execution**

## 4.4 Parallel Performance

Since there is loose dependency among threads, it is fairly easy to parallelize our analyzer in a *block decomposition* manner to asign a series of threads to a processing node. That is, if we have $m$ nodes $p_1, \ldots, p_m$, we decompose $N$ threads into $m$ sets $S_1, \ldots, S_m$ each of which has $N_k$ threads, namely $S_k = \{T_j \mid n(k-1) < j \leq n(k)\}$ where $n(k) = \sum_1^k N_k$. A node $p_k$ first performs the fast-forward simulation of $n(k-1)$ instructions, then does the differential simulation for threads in $S_k$ creating them, and continues it without thread creation until $N$ instructions are executed. Since fast-forwarding is much faster than out-of-order differential simulation, the execution time of $p_k$ is modeled as $N_k/a + (N - n(k))/b$ where $a$ and $b$ represent simulation speed with and without thread creation, if we may ignore the $O(\log N)$ factor of our algorithm. Then balancing the loads of nodes is achieved by determining $N_k$ to satisfy $N_1/a + (N - n(1))/b = \cdots = N_m/a + (N - n(m))/b$ which is solved as $N_k = r^{m-k} N_m$ where $r = 1 - a/b$ and $N_m = N(1-r)/(1-r^m)$.

By a few test runs of compress, we found $b = 20a$ approximately and evaluated all benchmarks with this value using an eight-node cluster of PCs equivalent to that shown in Section 4.1. The resulting parallel speedup shown in Figure 13 is quite satisfactory, except for hydro2d and m88ksim due to their peaks/plateaus of execution time discussed in the previous section. For other workloads, we achieved high speedup ranged from 6.33 (gcc) to 9.02 (perl). The reason of super-linear speedups of four workloads is the shrinkage of working set to result higher cache hit ratio.

## 5. RELATED WORK

Although most of traditional researches of WCET[11] aim at static analysis of program to find, for example, input data set to maximize its execution time, we can find several proposals using (cycle accurate) simulators to obtain a tight bound of WCET. For example, Engblom and Ermedahl proposed a combination of Implicit Path Enumeration Technique (IPET) and a trace driven microarchitecture simulator for a detailed analysis including instruction pipeline behavior across basic blocks[4]. Another example is found in the work of Burns et al.[3] which models instruction execution timing using a Petri-Net based simulation for superscalar processors.

On the other hand, our target WCID (or WCPD: Worst-Case Preemption Delay) problem had been attacked from the view point of the schedulability of interrupted/preempted processes (e.g. [2]). Then Lee et al.[5] pointed out the importance of the effect on caches, and proposed an analytical method to bound cache miss penalty due to interruptions. This approach was extended in two directions; to incorporate the cache pollution by preemptors[9, 13]; and to analysis more accurately using memory access trace obtained from the simulation or instrumented execution of a workload[7].

Although each of those work above has its own good feature, e.g. applicable without running[3, 4], taking into account multiple interruptions and/or preemptors[5, 7, 9, 13], they commonly have the problem on accuracy. That is, simulation based modeling inherently has a limitation to analyze *real* behavior of microarchitectural components, while cache-related delay analysis may underestimate the delay because it usually assumes some constant miss penalties.

Our analyzer, on the other hand, only has two sources of inaccuracy; microarchitectural model of SimpleScalar itself; and our assumption of interruption effect on CLMs, completely flushing, which causes some overestimation. Although the later inaccuracy is harder than the former to remove or reduce because of a huge space of possible pollution patterns, idea in previous work could be applicable to shrink the search space of worst-case pollution. Another limitation of our work is that so far only one interrupt may be inserted into the workload execution. Since a complete cycle count log of threads is easily obtained[9], a simple $O(N^2)$ dynamic programming based algorithm may be used to find WCID with multiple interruptions as suggested in [7]. The time complexity, however, must be significantly reduced in our future work because $N^2$ is larger than $10^{13}$ even with our relatively small workloads discussed in Section 4.

## 6. CONCLUSIONS

This paper describes an accurate and efficient algorithm to analyze WCID for one interrupt occurred in a workload using a cycle accurate simulator. The accuracy is assured because our analyzer performs simulation equivalent to that inserting interruption into all possible timings. The efficiency is achieved by our differential simulation technique by which a thread simulation takes place only in short durations in which its behavior is different from other thread. We also devised an efficient $O(\log N)$ algorithm to increment the cycle counts of sleeping threads using a binary tree of $N$ threads.

Our performance evaluation with SPEC CPU95 benchmarks proves the effectiveness of our differential simulation resulting fairly short execution time, 30 minutes or less, to analyze workloads of 5 million instructions which would cost up to 300 days without the technique. It is also confirmed that our algorithm works in $O(N(K + \log N))$ time assuring applicability to a wide range of applications. Moreover, a simple parallelization made our analyzer faster achieving up to 9.02-fold speedup on 8-node PC cluster.

[9]In fact, our analyzer outputs the cycle count each time an instruction retires in the execution of all active threads in addition to that of $T_0$.

Our most important future work is to devise an efficient algorithm to find WCID with multiple interruptions. We also plan to attack the problem of selective pollution by preemptors.

## Acknowledgments

This research work is partly supported by Grant-in-Aid of Scientific Research #17300015 of MEXT Japan, and 21th Century COE Program entitled "Intelligent Human Sensing" of MEXT Japan.

## 7. REFERENCES

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

[2] A. Burns, K. Tindel, and A. Wellings. Effective analysis for engineering real-time fixed priority schedulars. *IEEE Trans. Software Eng.*, 21(5):475–480, May 1995.

[3] F. Burns, K. Koelmans, and A. Yakovlev. WCET analysis of super-scalar processors using simulation with colored Petri nets. *Real-Time Systems*, 18(2/3):267–280, May 2000.

[4] J. Engblom and A. Ermedahl. Pipeline timing analysis using a trace-driven simulator. In *RTCSA'99*, pages 88–95, Dec. 1999.

[5] C.-G. Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Computers*, 47(6):700–713, June 1998.

[6] S. McFarling. Combining branch predictors. Technical Report WRL TN-36, DEC, June 1993.

[7] H. Miyamoto, S. Iiyama, H. Tomiyama, H. Takada, and H. Nakashima. An efficient search algorithm of worst-case cache flush timings. In *RTCSA 2005*, pages 45–52, Aug. 2005.

[8] H. Nakashima. An $O(\log N)$ algorithm to increment subarray members of an array of $N$ elements. Technical Report http://www.para.tutics.tut.ac.jp/TR/tree-add.pdf, Toyohashi U. Tech., 2006.

[9] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS 2003*, pages 201–206, Oct. 2003.

[10] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *ASPLOS-V*, pages 76–84, Oct. 1992.

[11] P. Puschner and A. Burns. A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2/3):115–128, May 2000.

[12] J. E. Smith. A study of branch prediction strategies. In *ISCA'81*, pages 135–148, May 1981.

[13] Y. Tan and V. Mooney. Integrated intra- and inter-task cache analysis for preemptive multi-tasking real-time systems. In *SCOPES 2004*, LNCS 3199, pages 182–199, Sept. 2004.