IPSJ Transactions on System LSI Design Methodology Vol. 1 33–47 (Aug. 2008)

Regular Paper

A Simulation-Based Analysis for Worst Case Delay of Single and Multiple Interruptions

Hiroshi Nakashima,[†1] Masahiro Konishi[†2]
and Takashi Nakada[†3]

This paper proposes an efficient method to analyze the *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. Furthermore, our CAS-based analyzer may have a post process to calculate the WCID for multiple F interrupts with $O(FN\sqrt{N}\log N)$ time complexity.

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 Ref. 13)), one of the most challenging themes is to find the upper bound of the delay caused by one (or more, sometimes) interruption which occurs in the execution of a program or a workload.

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 or more interruptions based on the simulation of a workload with a cycle accurate simulator (CAS). The first step is to find the delay caused by every possible interruption accurately by a CAS-based analysis 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 for this single-interrupt analysis, 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. Furthermore, we propose an $O(FN\sqrt{N}\log N)$ algorithm to find WCID with F interrupts from the cycle count log obtained from the CAS-based single-interrupt analyzer.

The rest of paper describes our WCID analyzers as follows. First the key idea of efficiency of the single-interrupt analyzer, *differential simulation*[6),11)] is introduced in Section 2. Our experimental results with our implementation based on SimpleScalar[1)] and SPEC CPU95 benchmarks are shown in Section 3. Then our $O(FN\sqrt{N}\log N)$ algorithm for WCID analysis with multiple interruptions is described in Section 4. After a brief discussion of related work in Section 5, we conclude this paper in Section 6.

†1 Kyoto University
†2 PFU Ltd.
†3 Nara Institute of Science and Technology

2. Differential Simulation

This section describes the key idea of our method for single-interrupt WCID analysis, *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*. For more formal and detailed discussion of the idea and its implementation, see Ref. 11).

2.1 Models of Processor and Interruption

Modern microprocessors with caches, branch predictors and an out-of-order instruction pipeline may 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). From the viewpoint of simulation, the architectural state at the completion of the instruction i_n, namely $A(n)$, can be obtained much more easily and quickly than the microarchitectural state, because it only requires instruction-level simulation which is usually about one hundred times as fast as out-of-order cycle accurate simulation. Thus the instruction-level simulation is sometimes called *fast-forwarding* and is often used for skipping leading instructions before the cycle accurate one for instructions with which an architect, for example, wants to measure the machine performance.

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 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. That is, a substate $C_{k,j}(t)$ for a cache set, for example, represents the tag and its validity of each way in the set and the recently-used ordering of the ways. The substates are independent of each other because a memory access, for example, affects (at most) only one substate causing a replacement, recently-used reordering, and so on.

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 apparent ones, can be approximated by an $O(N)$ simple algorithm presented in Ref. 5).

Note that a pair of executions with different interruption points, namely i_n and $i_{n'}$ $(n < n')$, should have different microarchitectural states at the fetch of $i_{n'+1}$, because the latter's has been just *flushed* while the former has something resulted from the execution of $i_{n+1}, \ldots, i_{n'}$. In general, these two executions may have

⋆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.

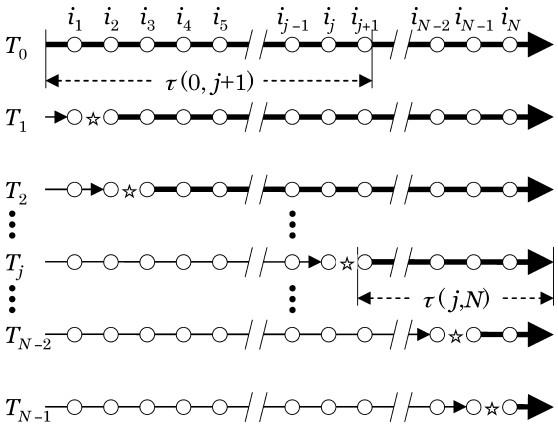


Fig. 1 Straightforward iterative simulation.

different microarchitectural states after the fetch of i_{n+1}.

2.2 Straightforward Iterative Simulation

It is easy to estimate single-interrupt WCID in a straightforward manner in which a workload is simulated iteratively varying interruption point as shown in **Fig. 1**. We define an instance of the workload execution of N instructions i_1, ..., 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, $\tau(0, j)$, each time the instruction i_j retires (topmost bold arrow in the figure). Note that $\tau(0, j+1) - 1$ is the worst case cycle count for the execution preceding the interruption at i_j.

(2) For each thread T_j $(1 \leq j < N)$ do the followings. First, obtain the architectural state at its beginning by an architectural *fast-forwarding* simulation (thin arrows in the figure). Then set the microarchitectural state 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 $\tau(j, N)$ (second and subsequent bold arrows in the figure).

(3) The WCET with one interrupt, namely τ_w, is given by;

$$\tau_w = \max_{0 \leq j < N} \{\tau(0, j+1) - 1 + \tau(j, N)\}$$

and WCID is given by $\tau_w - \tau(0, N)$.

It is obvious that this straightforward algorithm takes $O(N^2)$ time. For example, even for a tiny workload of $N = 10^6$, a CAS of 1 MIPS takes 0.5×10^6 second or about six days to simulate 0.5×10^{12} instructions.

2.3 Differential Out-of-order Simulation of Thread

The basic idea of the *differential simulation* is that the executions of adjacent threads are expected to behave *similarly*. That is, the simulation for a thread T_j may be omitted after its microarchitectural state becomes equivalent to that of T_{j-1}. Moreover, if the differences between the microarchitectural states of these threads are only in CLM, i.e., if the pipeline states of the threads are equivalent, we may omit the simulation for T_j until T_{j-1} *touches* the difference in CLM.

The expectation above is based on the following observations. First, the difference between the pipeline states of two threads at the fetch of i_{j+1} is found just in the fact that T_{j-1}'s pipeline, namely P^{j-1} may have i_j while T_j's pipeline P^j has nothing. Since the existence of i_j hardly affects the progress of the instruction i_{j+1} and its successors, both pipelines will become equivalent after i_j retires. Even if this hypothesis does not hold, two pipelines should become equivalent when they have instructions with long latency due to cache and branch prediction misses and then are made *sparse*. 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 misprediction occurs, and thus almost completely *forgets* its old memory. Therefore, after a certain small number of instructions are executed, the pipeline state of T_{j-1} and T_j should become equivalent. We call that T_{j-1} and T_j are *coherent* when their pipeline states are equivalent.

On the other hand, a CLM may have longer memory. For example, the instruction cache for T_{j-1}, namely C_{IC}^{j-1}, should have the block containing i_j after the fetch of it, while T_j's cache C_{IC}^{j} will not load the block some long time if i_{j+1} and successors reside in different blocks. However, when the both threads fetch the instruction i_j again, for example, the CLM substates for the block of

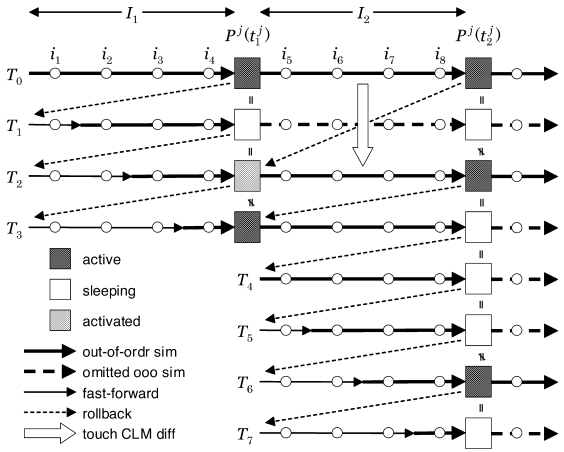


Fig. 2 Thread simulation in interval.

both threads should have the block and thus become equivalent. Note that the pipeline states of both threads should have been equivalent until the second fetch of i_j if the existence of the block in C_{IC}^{j-1} is the only difference in their CLM. Then the pipeline states become unequivalent because the fetch in T_{j-1} hits the cache while T_j misses, but they should become equivalent soon and again by, for example, a common branch misprediction.

Based on the observations above, we try to omit the simulation of a thread during it is coherent with its predecessor as follows. We compare the pipeline state of a thread with that of its predecessor periodically to examine if the thread may be made to *sleep*. More specifically, at the end of every *interval* of a predefined small number N_I of executed instructions, we examine the equivalence of the pipeline states as shown in **Fig. 2**. The figure shows the first two intervals, I_1 and I_2, of four instructions rather than eight in our 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 of SimpleScalar's pipeline model, saving its clock cycle t_1^0, the contents of architectural registers and pipeline state $P^0(t_1^0)$ for the next interval.

Table 1 Pipeline components of SimpleScalar.

queues	depth
inst. fetch queue	4
reg. 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	$1 \times$ (mult = 1/3; div = 19/20)
fp mult/div	$1 \times$ (mult = 1/4; 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.

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 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 in that both pipelines are equivalent, the thread T_1 *sleeps* until a CLM substate (e.g., a set of a cache) 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 of 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 2.4. The figure shows the case in which no differences are found in T_1's CLM and thus T_1 is kept asleep, but T_2 is *activated* by a 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 $P^0(t_1^0)$.

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

An important notice on this simulation is that the microarchitectural states of adjacent threads T_{j-1} and T_j must have a constant number of differences at the creation of T_j. Thus it is strongly expected that the differences in pipeline state will disappear by the execution of a constant number of instructions and those in CLM will do as well by a constant number of accesses. If this expectation is correct, the number of intervals in which T_j is active is bounded to some constant, and thus the time complexity of our simulation is $O(N)$ as far as the number of simulated instructions.

2.4 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 $C^j_{l,m}$ for T_j gives a result different from that of $C^k_{l,m}$ 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,

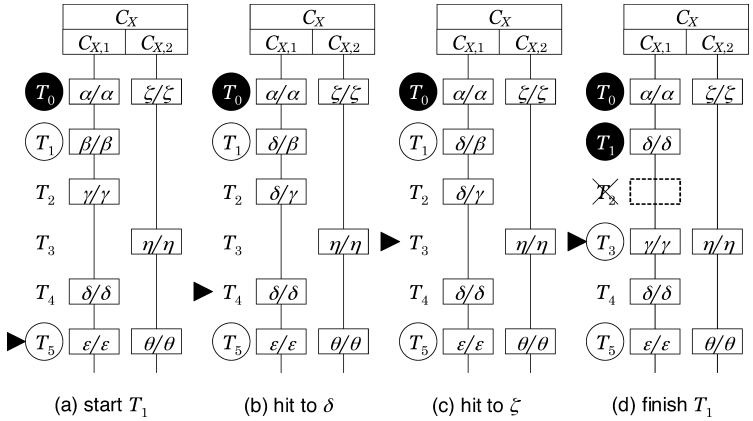


Fig. 3 Accesses to CLM substates.

Fig. 3 (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 values 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 γ/γ 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 circles), 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 δ. First T_1 examines its own substate and finds it results in a miss. Thus the *current* value of the substate is changed to δ but its *old* value remains unchanged. Then T_1 traverses all substate values, γ/γ and δ/δ, for threads dominated by it. Although the value γ/γ for T_2 (and T_3) is different from β/β for T_1, the access results affected to pipeline, e.g., the amount of miss-penalty, may be equivalent as we assume in this explanation. Thus the substate for T_2 is changed to δ/γ but T_2 (and T_3) is not activated. The substate δ/δ for T_4, however, has a different story because the hitting access to it should give a result different from the missing access to β/β. 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 triangle shown in Fig. 3 (b) which also shows the changes of substates.

Then T_1 accesses $C_{X,2}$ referring to its predecessor's substate ζ/ζ. Since the access hits to ζ, the substate remains unchanged ⋆1. Then it examines η/η for T_3 to find that coherence is broken again. Thus T_3 becomes the new activation candidate as shown in Fig. 3 (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 δ/β is changed to δ/δ because δ is now the *old* value for the next interval $I+1$; T_2's substate δ/γ 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 γ/γ is newly created and inserted copying T_2's old value γ, because T_3 must start from the beginning of I; and finally δ/δ for T_4 remains unchanged because it has not been modified ⋆2. 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 γ/γ and η/η.

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 α and β 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 $\beta{\rightarrow}\alpha$ where $\rightarrow$ represents recently-used precedence. For other threads invoked after the access with α, the value will be $\beta{\rightarrow}\perp$ or $\perp{\rightarrow}\perp$ 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 ⋆3.

Table 2 CLM of SimpleScalar.

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

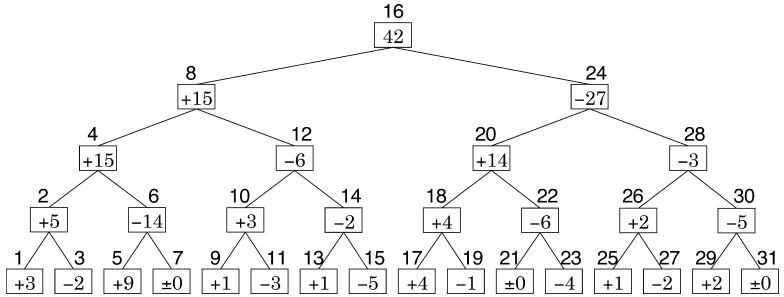
Table 2 shows all kinds of CLM which SimpleScalar supports and thus we implemented as well. Underlined configurations are SimpleScalar's defaults and are chosen for our experiment discussed in Section 3.

2.5 Maintenance of Cycle Counts of Sleeping Threads

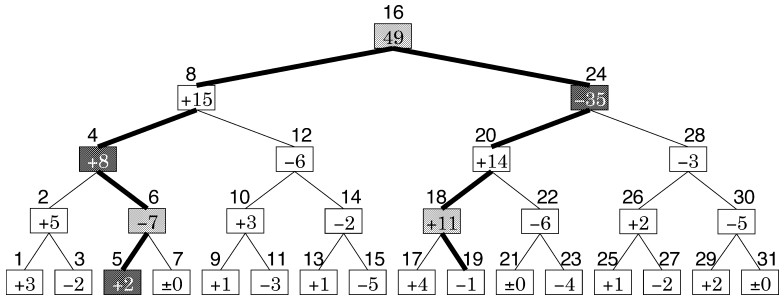
We have to not only simulate threads but also count cycles of thread executions to find WCID. Counting cycles of an active thread is of course obvious, but doing it for sleeping threads has a problem. The counting operation itself is fairly easy because we know the cycle when a thread went to sleep and 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 **Fig. 4** (a), cycle counts of threads, except for T_0, are kept in a binary tree whose nodes correspond to

⋆1 If the access makes the substate changed to, say κ, a new node κ/ζ would be *inserted* just below ζ/ζ for T_0 so that T_1 has its own substate.
⋆2 If it has been modified, we restore its old value into the current value to make it δ/δ.
⋆3 If we may ignore the effect of *false path* execution caused by branch mispredictions. This issue is discussed in Section 3.3.



(a) before addition



(b) after addition

Fig. 4 Maintenance of thread cycles using binary tree.

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 in $-14 + 15 + 15 + 42 = 58$.

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 Fig. 4 (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 δ to make it $(t^j + \delta) - t^{p(j)}$. In our example, n_6, n_{16} (with super-root assumption) and n_{18} satisfy 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 δ 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 condition.

The correctness of the algorithm shown above, i.e., the assurance that it keeps the invariant 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 $n_k \in S \leftrightarrow n_{p(k)} \in S$ holds for all n_k excluded from the upward path to the common ancestor (see Ref. 10) 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 maintenance is $O(N \log N)$ providing that the number of active threads at an interval is bounded by a constant. The calculation of the absolute cycle count is performed 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.

3. Experiment

3.1 Environment and Workload

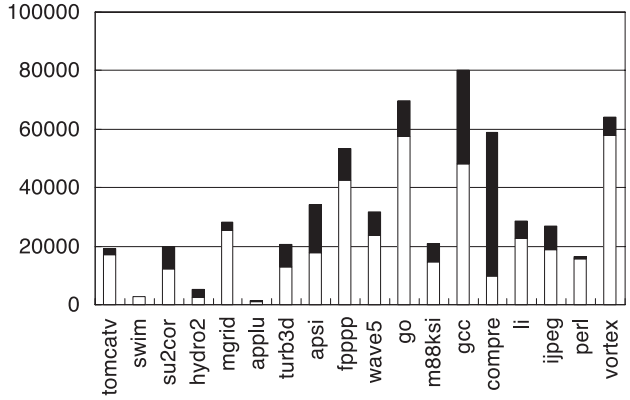
We implemented our single-interrupt WCID analyzer using SimpleScalar-3.0 as its base. The program is written in C as SimpleScalar is and is compiled by gcc 2.95.3 to run on an x86-based PC with Linux kernel 2.4.22. The performance is mesured using a 3 GHz Pentium-4 based PC with 1 GB memory. The target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

The workloads are all 18 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 applicability of our analysis method. Although our analysis is expected to be reasonably efficient, completely executing them would take too long time. In fact it should be significantly longer than ten days because the analysis speed is at least 4.5 times as slow as SimpleScalar's `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.

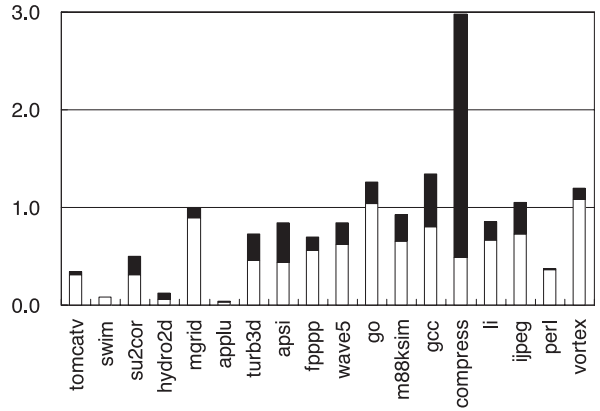
3.2 WCID Analysis Result

As the first result, WCID of each program is shown in **Fig. 5** 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 analyses. 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



(a) absolute delay [cycle]



(b) ratio to total cycle [%]

Fig. 5 Average and worst case interruption delays.

cycles. Thus, if we estimated the worst-case value from its medial and/or representative behavior, we could heavily underestimate the value resulting in a too optimistic estimation. In fact, as shown in **Fig. 6** 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 delay by a *run-through* investigation.

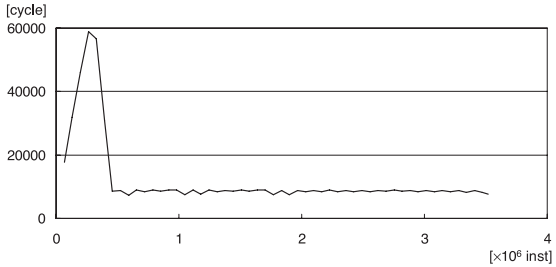


Fig. 6 Maximum delay caused by interruptions in each 64 K-instruction duration of compress.

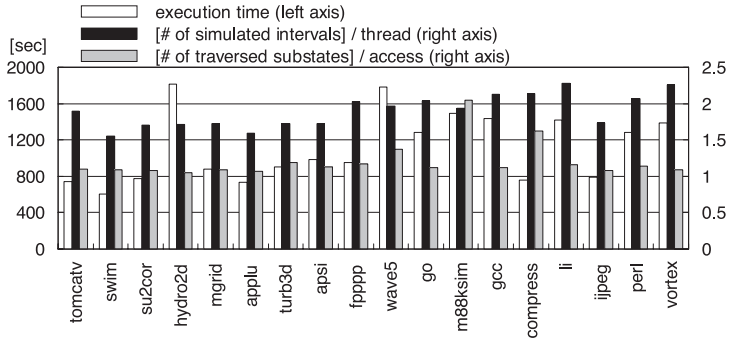


Fig. 7 Overall execution time, simulated intervals and traversed CLM substates.

3.3 Simulation Time and Related Performance Numbers

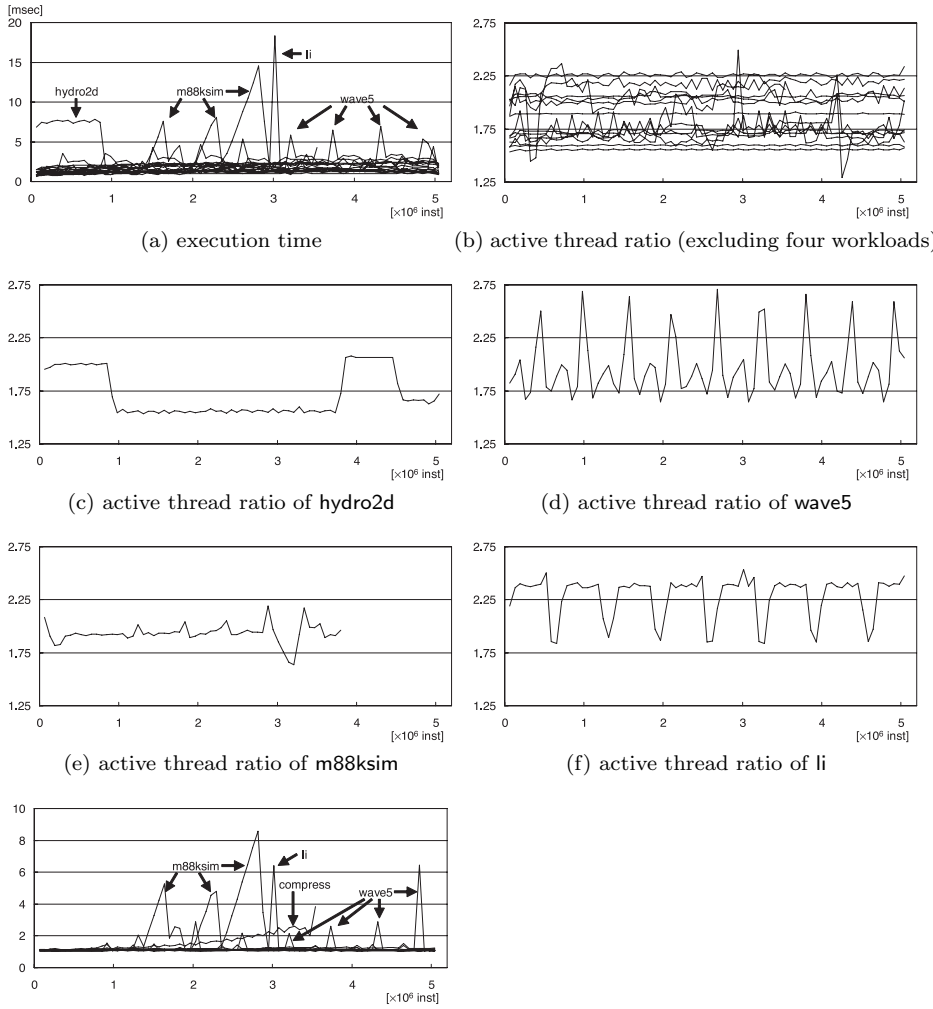
Since the most important expected feature of our WCID analyzer is its efficiency achieved by the differential simulation, measuring the performance numbers of its execution time is essential. First we show basic overall numbers in **Fig. 7** in which wall-clock execution times (white bars) are illustrated 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 substate values 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 12.5×10^{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 25×10^{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 traversals 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.

Although 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 **Fig. 8**. The execution times shown in Fig. 8 (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 peaks and plateau, however, look reflecting the computation 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 Fig. 8 (b) to (f), and substate traversals shown in (g). Figure 8 (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 Fig. 8 (g) matches and thus clearly explains the peaks in Fig. 8 (a). A possibly bad news that Fig. 8 (g) tells us is that the number in compress looks steadily growing, but this may simply show the first portion of a long but transient phase shift which could appear clearly if compress had much more instructions executed.



(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

(g) Traversed CLM substate ratio

Fig. 8 Execution time, active thread ratio and traversied CLM substate ratio in each interval.

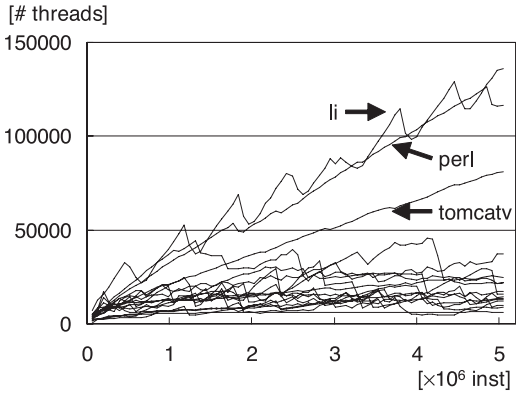


Fig. 9 Number of sleeping threads in each interval.

Finally, we measured the number of sleeping threads in each interval, which is shown in **Fig. 9**. The figure clearly shows that the number is proportional 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 proportional to the number of them.

These results also reveal that our implementation needs $O(N)$ memory space not only for maintaining thread cycle counts using the binary tree discussed in Section 2.5 but also for keeping the contexts of sleeping threads. However, although a context is significantly larger than a node of the tree, 68 byte versus 8 byte in our implementation, the small ratio to N, about 2.7% even for li, softens the impact from the size of contexts.

The other concern arisen from the results should be that the $O(N)$ number of sleeping threads means that the total number of CLM substate values in the linked lists discussed in Section 2.4 should be also $O(N)$. In fact, we observed the numbers for three benchmarks discussed above are proportional to N, about 3.2% of N for li and perl. Since a sleeping thread should have at least one CLM substate value of its own, this observation looks unsurprising. The impact to the memory requirement is heavier than that from the context due to, for example, $28w + 28$ byte consumption by a set of w-way set associative cache in

our implementation, but still acceptable because 3.2% ratio and $w = 4$ gives us an expected value of 4.5 byte per thread.

However, it contradicts the discussion in Section 2.4 where we stated that the number of thread-own values of a CLM substate should be bounded to a constant because a value node for a thread is removed when it becomes equivalent with its predecessor. This contradiction can be explained by the effect of branch mispredictions and the execution of *false paths* caused by the mispredictions. Let us assume a workload and a loop in it. Also let us assume the conditional branch for the loop termination is mispredicted on the first execution after an interruption occurring in the loop execution. Since the behavior of the pipeline on the misprediction should depend on the number of instructions *on-the-fly* in the pipeline, the behavior could also depend on the distance from the interruption point to the branch.

For example, if the interruption point is near to the branch, the misprediction will be recognized quickly in the back-end of the pipeline and thus only one block on the false path is loaded to instruction cache. On the other hand, in the case of large distance, two blocks are loaded because the execution of the branch is delayed by other preceding instructions. Thus now we have two groups of threads; one is *far* group whose members have loaded two blocks from the false path to instruction cache, while the other *near* ones have done only one block. Since both blocks will not be accessed again until the loop termination, the second block in instruction cache is the far group's own and stays in cache if it is large enough. Then if the loop iterates many times enough, we will have a large number of alternating thread groups of far and near and each group disagree with its predecessor group with the second cache block causing a list of CLM substate values whose length is proportional to the number of the loop iterations.

Note that the scenario above does not contradict our observation that the number of substate traversals per access is constant. Of course the long list will be traversed after the loop termination, but the traversal should shrink the list because the second cache block for every thread becomes having the instructions following the loop commonly.

From those results shown in Figs. 8 and 9 and discussions above, 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. As of the spatial complexity, we need $O(N)$ space but the proportionality constant is acceptably small, ten or so byte per instruction executed in a workload.

4. WCID Analysis with Multiple Interrupts

This section describes an algorithm to calculate WCID with F interrupts from the output of the single-interrupt CAS-based analyzer. Before discussing our $O(FN\sqrt{N}\log N)$ algorithm, a simple $O(FN^2)$ one is shown as the base.

Let $\tau(j,k)$ $(j < k)$ be the cycle count at the retirement of the instruction i_k in the thread T_j which starts from the interruption at i_j. If the execution of the thread T_j is interrupted again at i_k, i.e. after the retirement of i_k and before that of i_{k+1}, the WCET between these two interrupts should be $\tau(j,k+1)-1$. Therefore, the WCET with F interrupts, namely $W(F)$, for a workload with N executed instructions is given by the following equation.

$$W(j,f) = \max\left\{ \sum_{l=0}^{f} \tau(k_l, k_{l+1}+1) - 1 \;\middle|\; k_0 = j,\ k_{f+1} = N-1,\ k_i \le k_{i+1} \right\} \quad (1)$$

$$W(F) = W(0,F) + 1 \quad (2)$$

Similar to the formulation in Ref. 8), the equation (1) can be rewritten as follows because $\tau(k_l, k_{l+1}+1)$ is independent from any other $\tau(k_{l'}, k_{l'+1}+1)$.

$$W(j,0) = \tau(j,N) - 1$$

$$W(j,f) = \max_{j \le k < N} \{\tau(j,k+1) - 1 + W(k,f-1)\} \quad (3)$$

From the recurrence (3) above, we have the following $O(FN^2)$ algorithm based on dynamic programming technique.

```
for j = 0 to N do W[j][0] ← τ(j,N) − 1;
for f = 1 to F do begin
  for j = 0 to N − 1 do begin
    w ← 0;
    for k = j to N − 1 do  w ← max(w, τ(j,k+1)−1+W[k][f−1]);
    W[j][f] ← w;
  end
```

end

Our single-interrupt WCID analyzer outputs the cycle count of each thread T_j not only at the end of the simulation (i.e., $\tau(j, N)$), but also each time an instruction is retired during the thread is actively simulated (i.e., $\tau(j, k)$ for each k s.t. i_k is in an active interval of T_j) ⋆1. Thus if it is allowed to store $\tau(j, k)$ in a two-dimensional array paying $O(N^2)$ spatial complexity cost, the array is easily filled with the cycle count log interpolating $\tau(j', k')$ for the thread $T_{j'}$ which slept in the interval of $i_{k'}$ by the count of its dominator. Otherwise, i.e., $O(N^2)$ is too large ⋆2, the log should be repeatedly scanned and $\tau(j', k')$ for a sleeping thread should be dynamically calculated by the algorithm discussed in Section 2.5 resulting in $O(FN^2 \log N)$ temporal complexity.

Anyway, $O(FN^2)$ or $O(FN^2 \log N)$ temporal complexity is too large and thus should be significantly reduced. The key idea for the complexity reduction is that a part of the equation (3) for a sleeping thread $T_{j'}$ may be calculated from that for the other thread. More specifically, if $T_{j'}$ sleeps in a duration for instructions in $\{i_k \,|\, l < k \le l'\}$ and is dominated by T_j, the following equation holds for each k such that $l < k \le l'$.

$$\tau(j', k) = \tau(j, k) + (\tau(j', l) - \tau(j, l))$$

Therefore the corresponding part of the equation (3) may be rewritten as follows.

$$\begin{aligned} W(j', f, l, l') &= \max_{l \le k < l'} \{\tau(j', k+1) - 1 + W(k, f-1)\} \\ &= \max_{l \le k < l'} \{\tau(j, k+1) - 1 + (\tau(j', l) - \tau(j, l)) + W(k, f-1)\} \\ &= \max_{l \le k < l'} \{\tau(j, k+1) - 1 + W(k, f-1)\} + (\tau(j', l) - \tau(j, l)) \\ &= W(j, f, l, l') + (\tau(j', l) - \tau(j, l)) \end{aligned} \tag{4}$$

The equation (4) above means that we may skip the part of the innermost loop, from $k = l$ to $k = l' - 1$, to calculate $W(j', f)$ of the $O(FN^2)$ algorithm, if $T_{j'}$ slept in this duration and we have known the value for T_j, i.e., $W(j, f, l, l')$. To accomplish the complexity reduction by this calculation skipping, we decompose N instructions into n segments of a fixed number m of instructions, $(0, m]$, $(m, 2m]$, $\ldots$, $((n-1)m, nm]$ where $nm = N$. Then we calculate $W(j, f, lm, (l+1)m)$ for each l such that $0 \le l < n$ using the equation (4) to obtain $W(j, f)$ as follows.

$$\begin{aligned} W'(j, f, lm, (l+1)m) &= W(j-1, f, lm, (l+1)m) + \\ &\qquad (\tau(j, lm) - \tau(j-1, lm)) \\ W''(j, f, lm, (l+1)m) &= \max_{lm \le k < (l+1)m} \{\tau(j, k+1) - 1 + W(k, f-1)\} \\ W(j, f, lm, (l+1)m) &= \begin{cases} 0 & \text{if } j \ge (l+1)m \\ W'(j, f, lm, (l+1)m) & \\ \qquad \text{if } T_j \text{ slept in the duration} & \\ \qquad \text{for } \{i_k \,|\, lm < k \le (l+1)m\} & \\ W''(j, f, lm, (l+1)m) & \\ \qquad \text{otherwise} & \end{cases} \end{aligned} \tag{5}$$

$$W(j, f) = \max_{0 \le l < n} \{W(j, f, lm, (l+1)m)\} \tag{6}$$

Our evaluation shown in Section 3.3 assures that threads sleep almost always and are active only in a small constant number of intervals, about two intervals or 16 instructions. Therefore, it is strongly expected that the calculation of the third case of the equation (5) is required only for a small constant times and thus $W(j, f)$ is obtained in $O(n + m)$ time for each j. Under the condition of $N = nm$, the time complexity $O(n + m)$ is minimized by setting $n = c\sqrt{N}$ and $m = (1/c)\sqrt{N}$ with some constant c to result in $O(\sqrt{N})$ complexity. Thus we have the following algorithm to calculate WCID with F interrupts.

```
n ← c√N; m ← N/n;
for j = 0 to N do  W[j][0] ← τ(j, N) − 1;
for f = 1 to F do begin
  for j = 0 to N do  W[j][f] ← 0;
  for l = 0 to n − 1 do begin
    for j = 0 to (l + 1)m − 1 do begin
      if T_j sleeps in the duration for {i_k | lm < k ≤ (l + 1)m} then
        w ← w + (τ(j, lm) − τ(j − 1, lm));
      else begin
        w ← 0;
        for k = max(lm, j) to (l + 1)m − 1 do
```

⋆1 The execution time discussed in Section 3 includes the cost for this per-instruction cycle count logging.

⋆2 It should be too large because even our small workload with 5×10^6 instructions needs a huge array of 12.5×10^{12} element or 50 TB if each element is represented in a 4-byte integer.

```
      w ← max(w, τ(j, k+1) − 1 + W[k][f − 1]);
    end
    W[j][f] ← max(W[j][f], w);
  end
 end
end
WCID ← W[0][F] + 1 − τ(0, N);
```

The algorithm above works in $O(FN\sqrt{N})$ time if each $\tau(j,k)$ referred in the algorithm is obtained in a constant time. This is accomplished by a preprocessing of the cycle count log from the single-interrupt WCID analyzer to store the values of required $\tau(j,k)$ in a two-dimensional array of $N \times n$ which is partially three-dimensional for the reference from the most inner k loop. However, the spatial complexity $O(N\sqrt{N})$ may be still too large because $N = 5 \times 10^6$ and $c = 1$ result in about 1.1×10^{10} elements or 45 GB with a 4-byte integer for each element.

Therefore, it will be necessary to scan the log repeatedly for each f and to calculate required $\tau(j,k)$ also repeatedly by the algorithm in Section 2.5 for the duration in which T_j slept. This means we have to pay $O(\log N)$ cost for each l and thus the time complexity of the algorithm is $O(FN\sqrt{N}\log N)$.

The implementation of the $O(FN\sqrt{N}\log N)$ algorithm is on the way and thus our urgent future work. This work includes finding the optimal setting of c and the evaluation of WCID and the performance of the algorithm.

5. Related Work

Although most of traditional researches of WCET [13] 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 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., Ref. 2)). Then Lee, et al. [7] 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 [12),14)]; and to analyze more accurately using memory access trace obtained from the simulation or instrumented execution of a workload [8].

Although each of those work above has its own good feature, e.g., applicable without running [3),4)], efficiently handling multiple interruptions and/or preemptors [7),8),12),14)], 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 analyses may underestimate the delay because they usually assume 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 CLM, 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 the worst-case pollution.

Another problem of our work could be found in the efficiency issue especially for the multiple interruption analysis. That is the time complexity $O(FN\sqrt{N}\log N)$ of our algorithm is significantly larger than that of cache-related delay analyses, e.g., $O(FN)$ in Ref. 9) and Ref. 5). This is mainly due to that our algorithm takes care of each sleeping thread each time it processed m (or $\sqrt{N}/c$) instructions, while the second case of the equation (6) holds throughout the sequence of segments in which the thread continuously sleeps. Therefore, our algorithms could be improved further if we find a more sophisticated means to skip the calculation of $W(j, f, lm, (l+1)m)$.

6. Conclusions

This paper describes an accurate and efficient algorithm to analyze WCID for one interrupt which occurs 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 in 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.

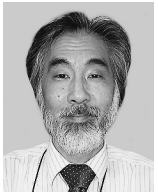
As for the WCID analysis with multiple interruptions, we proposed an algorithm of $O(FN\sqrt{N}\log N)$ time complexity. The implementation, evaluation and complexity reduction of the algorithm are our most important and urgent future work. We also plan to attack the problem of selective pollution by preemptors.

References

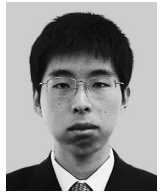
1) Austin, T., Larson, E. and Ernst, D.: SimpleScalar: An Infrastructure for Computer System Modeling, *Computer*, Vol.35, No.2, pp.59–67 (2002).
2) Burns, A., Tindel, K. and Wellings, A.: Effective Analysis for Engineering Real-Time Fixed Priority Schedulars, *IEEE Trans.Software Eng.*, Vol.21, No.5, pp.475–480 (1995).
3) Burns, F., Koelmans, K. and Yakovlev, A.: WCET Analysis of Super-scalar Processors Using Simulation with Colored Petri Nets, *Real-Time Systems*, Vol.18, No.2/3, pp.267–280 (2000).
4) Engblom, J. and Ermedahl, A.: Pipeline Timing Analysis Using a Trace-Driven Simulator, *RTCSA'99*, pp.88–95 (1999).
5) Konishi, M., Nakada, T., Tsumra, T., Nakashima, H. and Takada, H.: An Efficient Analysis of Worst Case Flush Timings for Branch Predictors, *IPSJ Trans. ACS*, Vol.48, No.SIG8 (ACS18), pp.127–140 (2007).
6) Konishi, M., Nakada, T., Tsumura, T. and Nakashima, H.: Measuring Worst-Case Performance of Microprocessor by Interruption with Omitting Redundant Execution, *IPSJ Trans. ACS*, Vol.47, No.SIG12 (ACS15), pp.159–170 (2006). In Japanese.
7) Lee, C.-G., et al.: Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling, *IEEE Trans. Computers*, Vol.47, No.6, pp.700–713 (1998).
8) Miyamoto, H., Iiyama, S., Tomiyama, H., Takada, H. and Nakashima, H.: An Efficient Search Algorithm of Worst-Case Cache Flush Timings, *RTCSA 2005*, pp.45–52 (2005).
9) Miyamoto, H., Iiyama, S., Tomiyama, H., Takada, H. and Nakashima, H.: An Efficient Search Algorithm of Worst-Case Cache Flush Timings, *IPSJ Trans. ACS*, Vol.46, No.SIG 16 (ACS12), pp.85–94 (2005). In Japanese.
10) Nakashima, H.: An $O(\log N)$ Algorithm to Increment Subarray Members of an Array of N Elements, Technical Report http://www.para.media.kyoto-u.ac.jp/TR/tree-add.pdf, Kyoto University (2006).
11) Nakashima, H., Konishi, M. and Nakada, T.: An Accurate and Efficient Simulation-Based Analysis for Worst Case Interruption Delay, *CASES 2006*, pp.2–12 (2006).
12) Negi, H.S., Mitra, T. and Roychoudhury, A.: Accurate Estimation of Cache-Related Preemption Delay, *CODES+ISSS 2003*, pp.201–206 (2003).
13) Puschner, P. and Burns, A.: A Review of Worst-Case Execution-Time Analysis, *Real-Time Systems*, Vol.18, No.2/3, pp.115–128 (2000).
14) Tan, Y. and Mooney, V.: Integrated Intra- and Inter-Task Cache Analysis for Preemptive Multi-tasking Real-Time Systems, *SCOPES 2004*, LNCS 3199, pp.182–199 (2004).

(Received December 25, 2007)
(Revised March 17, 2008)
(Accepted May 1, 2008)
(Released August 27, 2008)

(Recommended by Associate Editor: *Hideharu Amano*)



Hiroshi Nakashima received his M.E. and Ph.D. from Kyoto University in 1981 and 1991 respectively, and was engaged in research on inference systems with Mitsubishi Electric Corporation from 1981. He became an associate professor at Kyoto University in 1992, a professor at Toyohashi University of Technology in 1997, and a professor at Kyoto University in 2006. His current research interests are the architecture of parallel processing systems and the implementation of programming languages. He received the Motooka award in 1988 and the Sakai award in 1993. He is a member of IPSJ, IEEE-CS, ACM, ALP and TUG.



Masahiro Konishi received his M.E. degree from Toyohashi University of Technology in 2006 and joined PFU Ltd. that year. As a student, he engaded in research work on microprocessor simulators.

Takashi Nakada received his M.E. degree in 2004 and his Ph.D. in 2007 from Toyohashi University of Technology. He joined Nara Institute of Science and Technology in 2007 as an assistant professor. His current research interests are computer architecture and related simulation technologies. He is a member of IPSJ.