

Regular Paper

An Efficient Analysis of Worst Case Flush Timings for Branch Predictors

MASAHIRO KONISHI,^{†1} TAKASHI NAKADA,^{†2}
TOMOAKI TSUMURA,^{†3} HIROSHI NAKASHIMA^{†4}
and HIROAKI TAKADA^{†5}

This paper proposes an efficient algorithm to find the *worst case flush timings* for a given program with respect to the number of branch mispredictions. We first give a basic algorithm based on dynamic programming which takes $O(N^2F)$ computation time for a program with N conditional branches and F flush timings. We then show it can be improved to achieve a computation time of approximately $O(NF)$ for practical programs with its proof obtained through an evaluation with SPEC CPU95 benchmarks.

1. Introduction

For real-time system design, *worst case execution time* (WCET) analysis of a program is indispensable to verify and/or to ensure that the program completes its work within a given temporal restriction. In this technological field, a variation of the problem to find the upper bound of the performance degradation caused by an interruption/preemption is considered as one of the greatest challenges.

This *worst case interruption/preemption delay* is determined by two major factors. One is *external* delay corresponding to the time spent by preempting processes and the operating system, and this is strongly related to the schedulability problem for the interrupted/preempted process. The target of our research, though, is the other factor, *internal* delay, which the interrupted/preempted process incurs during its execution due to the loss of locality.

Since the efficiency of modern processors depends on the various temporal/

spatial localities of program execution, a process switching inevitably causes significant performance damage to the interrupted/preempted process which loses its own contents of caches, branch predictors and instruction pipeline. Previous work mainly focused on the damage to caches^{11),15),18),22)} to analyze how many misses are added by one or more interruptions/preemptions and to find the set of worst case timings of these.

Although these *worst case flush timing* (WCFT) analyses give a good approximation of the delay because caches suffer the most critical damage, we cannot ignore the damage on the other important hardware component, branch predictors, which we aim at in this paper. Branch predictors have been almost completely neglected in previous WCFT work because their mechanism is more complicated than caches. For example, the worst case cache state for any program may be obviously defined as the fully invalidated one, while that of a 2-bit counter of a basic bimodal predictor²¹⁾ depends on branches whose actions are predicted by the counter. The other source of analysis difficulty is that an arbitrary number of past branches may affect the counter state, while a cache set state is determined by a fixed number of accesses to it, equal to its associativity.

These difficulties also imply that the delay caused by a flush of predictor tables is as significant as, or may be more significant than, that of caches. For a cache, the number of additional misses caused by a flush is obviously bounded by its number of blocks (or lines) and is often less than that because it may have unuseful blocks. On the other hand, a flush of a 2-bit counter may affect the prediction accuracy of an arbitrary number of branches following it and thus the additional misses may be larger than the number of counters in a predictor table.

Our contribution to this hard WCFT problem for branch predictors is twofold. First we show that a dynamic programming technique, which solved the WCFT problem of caches¹⁵⁾, is also applicable to the problem for bimodal predictors to obtain an $O(N^2F)$ algorithm. Secondly and more importantly, we improve the algorithm to achieve $O(NF)$ time complexity in a practical sense and exhibit its efficiency through our evaluation with SPEC CPU95 benchmarks. In this paper, these two contributions are discussed within the following structure. First, previous related research on WCET and WCFT is summarized in Section 2 to show the background of our work. Section 3 gives a few definitions and notation to formulate the problem and algorithms. Section 4 is the heart of this paper where two algorithms, basic $O(N^2F)$ and improved $O(NF)$, are shown. Section 5 presents performance numbers for the two algorithms with regard to SPEC CPU95 benchmarks. After briefly dis-

†1 Toyohashi University of Technology
Presently with PFU Ltd.

†2 Nara Institute of Science and Technology

†3 Nagoya Institute of Technology

†4 Kyoto University

†5 Nagoya University

cussing the applicability of our algorithms to other predictors in Section 6, we conclude in Section 7.

2. Related Work

Traditionally, the WCET problem has been attacked by an analysis of the *logical* behavior of a program to find, for example, the input data set that maximizes the number of operations/instructions executed in the program²⁰). Although the problem is not decidable in general, a large number of *practical* programs for real-time systems can be successfully analyzed by sophisticated methods often relatively efficiently.

On the other hand, it is harder to analyze the *physical* behavior of a program running on a given computational environment. For example, a data set A to maximize the number of executed instructions could result in a shorter execution time than another set B , the execution with which has fewer executed instructions but more cache misses than A . The problem is made more complicated by other mechanisms of modern processors, such as instruction pipeline, out-of-order and/or parallel instruction execution, and branch prediction combined with speculative execution.

This physical behavior analysis to predict WCET, however, is not the hardest problem because it is widely believed that an execution with the worst case data set on a real or simulated computational environment gives a good approximation of WCET. Even when this straightforward method does not work, various analytical methods would successfully give you a safe and tight upper bound of WCET with caches (e.g. 6), 8), 23)), branch predictors (e.g. 2), 4)), and/or their combination with pipelined and speculative execution^{3),5),12}).

The first proposal to incorporate branch predictors into WCET analysis was presented by Colin and Puaut⁴). They focused on the *residentiality* of a branch in a *branch target buffer* (BTB) and examined whether a branch must reside in a BTB or may not, using a technique similar to those for caches²³). Although they modeled that a BTB entry also has a 2-bit counter to predict the direction of a conditional branch, they simply assumed that the direction of a branch to iterate a loop is correctly predicted except for the loop termination if it resides in BTB, while that for if-then-else is always mispredicted.

This simple but rough modeling was partially refined by Bate and Reutemann²), especially for nested loops for which they took into account the cases with small iteration counts and predictors using *global history* such as *gselect*¹⁹). Regarding if-then-else constructs, they pointed out the scheme of Colin and Puaut may cause underestimation, but they still adopted the always-mispredict assumption. They also proposed a method to combine the analyses of mispre-

dition and pipeline timing³), while the other effect, instruction cache pollution by misprediction, is discussed by Li et al.¹²).

The behavior of a program in a *real* execution environment, in which an operating system switches processes according to a scheduling policy, is much harder to analyze. As briefly discussed in the Introduction, a process preempted by preemptors and the operating system, and *external* delay time spent by preemptors and the operating system, and *internal* delay caused by the loss of execution locality. After the former delay issue was explored as a schedulability problem, the latter delay was incorporated by focusing on caches by Lee et al.¹¹).

In the literature, they formalized *cache-related preemption delay* as the sum of the number of *useful* cache blocks at each preemption point. By a data-flow analysis of a target program, each point of the program execution is labeled with the number of useful cache blocks which will afterward be accessed. The point having the highest cost, the maximum number of useful blocks, is then chosen as the worst-case preemption point and its worst-case visit count is multiplied by the cost to obtain the total cost of multiple preemptions. If the predefined number of preemptions is larger than the visit count, the points with the second highest cost and succeeding costs are chosen up to the preemption count. Finally, the preemption costs are combined with the external delays to determine the worst-case delay calculated by an integer linear programming technique.

Although this method gives us a safe upper bound of the preemption delay, it is heavily overestimated because the cost is incurred each time the point is visited by an execution of the process. Thus if a point in a loop has the maximum cost and the loop iteration count is large enough, the total cost of F preemptions will be F times the maximum cost which is obviously and heavily overestimated since a series of contiguous preemption results in a much smaller cost. Other proposals aim at refining this scheme by taking account of preempting processes^{18),22}), but they still stand on Lee's assumption. It is also noteworthy that, to our knowledge, no proposals have been made to combine Lee's scheme and branch predictors although this would be technically feasible.

Another approach to the preemption delay problem is to analyze a *workload*—i.e., the combination of a program and its (worst-case) input data set—rather than to analyze the program solely and statically. While this approach should be too natural and trivial for typical WCET problems, it is still challenging for the preemption delay problem because we have a huge search space from which we have to find out the worst-case preemption points. In the first proposal using this approach, Miyamoto et al.¹⁵) used it to find the *worst*

case flush timings (WCFT) of direct-map caches for a process preempted F times. That is, WCFT is the set of preemption timings that maximizes the total cache misses in the execution of the process.

They also gave the following formula to obtain WCFT from a trace of N memory accesses:

$$\Gamma(i, f) = \max_{i \leq j \leq N} \{C(i, j) + \Gamma(j, f - 1)\}$$

where $C(i, j)$ is the number of useful cache blocks loaded after the i -th memory access and flushed just after the j -th access, and which are referred to by the $(j + 1)$ -th and subsequent memory accesses, and $\Gamma(i, f)$ is the worst-case total useful blocks lost by f flushes which occur after the flush at the i -th access. This formula strongly suggests that the problem is solved by a dynamic programming technique in time $O(N^2F)$. Their more recent report¹⁴⁾ showed a better algorithm whose time complexity is $O(NF)$ if we assume the average gap between two accesses to a cache block is a constant independent of N . This improvement is based on the fact that the effect of an access to a direct-map cache is *nullified* by the following access to the same cache block because what resides in the block is determined only by the second access regardless of the first one.

Our WCFT analysis is, to our knowledge, the first attempt to analyze the worst-case preempted behavior of branch predictors. Although our method is inspired by Miyamoto's, predictors are much harder to analyze than caches for the following reasons. First, a cache block has the apparent worst-case *invalid* state, while four states of a bimodal predictor's 2-bit counter are equally meaningful for branches whose actions are predicted by the counter. Thus the worst-case value of a counter at a preemption point is a function of all the branches executed after the point, whose straightforward evaluation should take $O(N)$ time with N executed branches rather than $O(1)$ in the case of caches. Second, a cache block *forgets* its past after a constant number of accesses but a 2-bit counter may *remember* the action of a branch executed long ago. Since the forgetfulness of caches is the basis of Miyamoto's $O(NF)$ algorithm, the strong retention of 2-bit counters is a substantial barrier to efficient solutions.

3. Definitions and Notation

First we define the bimodal branch predictors we aim to analyze as follows (**Fig. 1**).

Definition 1 Let b_1, \dots, b_N be conditional branches having indices $1 \leq i \leq N$ executed in a process to be analyzed in this order, P be a set of 2-bit

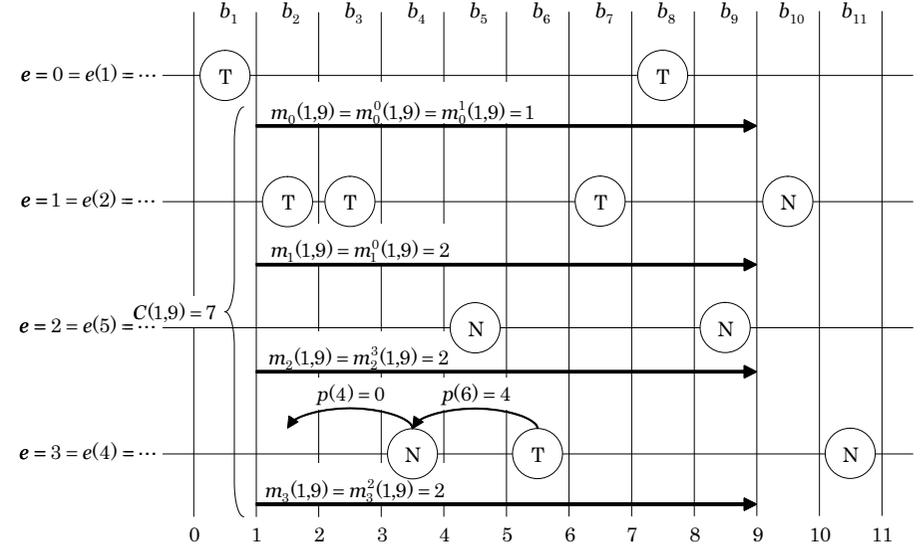


Fig. 1 Definitions and notation.

saturating counters $c[0], \dots, c[|P| - 1]$, and e be the function of branch index i to associate a branch b_i to a counter $c[e(i)]$. A branch b_i is predicted to be *taken* (or *not-taken*) if $c[e(i)] \geq 2$ (or $c[e(i)] < 2$). After the prediction, $c[e(i)]$ is *atomically* updated to $\min(c[e(i)]+1, 3)$ (or $\max(c[e(i)]-1, 0)$) if b_i is eventually taken (or is not taken) as denoted by $b_i \mapsto T$ (or $b_i \mapsto N$). A taken (or not-taken) branch b_i is *mispredicted* if and only if $c[e(i)] < 2$ (or $c[e(i)] \geq 2$). The function $p(i)$ gives the index of the immediate predecessor of b_i which shares the counter $c[e(i)]$, or 0 if b_i is the first branch referring to $c[e(i)]$. That is,

$$p(i) = \max(\{j \mid e(i) = e(j), j < i\} \cup \{0\})$$

In the usual configuration, $|P|$ is a power of 2 and $e(i)$ is calculated by extracting the low $\log |P|$ bits of the address where b_i resides. The *atomic* update of the counters means that the update of $c[e(i)]$ by b_i is immediately reflected in the predictions of succeeding branches b_{i+1}, b_{i+2}, \dots if they refer to the counter $c[e(i)]$. Although this assumption is not always correct in real implementations because a branch could refer to a counter before its predecessor updates the counter, this racing seldom occurs in real executions on a simple pipeline for embedded processors and almost never affects the prediction.

Next we define the *worst case flush timings* (WCFT) and related costs as follows.

Definition 2 Let $m_e^\gamma(i, j)$ be the number of mispredictions of the branches in $\{b_k \mid i < k \leq j, e(k) = e\}$ assuming $c[e] = \gamma$ when b_i is completed (or at the beginning if $i = 0$). If the execution is preempted somewhere between b_i and b_{i+1} (or before b_1 if $i = 0$) and thus $c[e]$ is *flushed* to have an arbitrary value, the worst case number of mispredictions of the branches is given by $m_e(i, j) = \max_{0 \leq \gamma \leq 3} \{m_e^\gamma(i, j)\}$. Thus, the *flush cost* at i which b_{i+1}, \dots, b_j suffer is defined as $C(i, j) = \sum_e m_e(i, j)$. The following equation gives the worst case cost caused by the flush at i and subsequent f flushes.

$$\Gamma(i, f) = \max_{j_1, \dots, j_f} \left\{ \sum_{k=0}^f C(j_k, j_{k+1}) \mid j_0 = i, j_{f+1} = N, j_k \leq j_{k+1} \right\} \quad (1)$$

Thus, $\Gamma(0, F)$ gives the worst case cost of F flushes, and WCFT is the set of branch indices j_1, \dots, j_F which maximizes $\sum C(j_k, j_{k+1})$ to define $\Gamma(0, F)$.

4. WCFT Algorithms

4.1 Dynamic Programming Algorithm

Similar to the formulation in [15], equation (1) can be rewritten as follows because $C(j_k, j_{k+1})$ may be calculated independently from any other $C(j_{k'}, j_{k'+1})$.

$$\Gamma(i, f) = \max_{i \leq j \leq N} \{C(i, j) + \Gamma(j, f - 1)\} \quad (2)$$

From this recurrence and the initial condition $\Gamma(i, 0) = C(i, N)$ derived from (1), the following algorithm, namely *DPWCFT* is obtained by a dynamic programming (DP) technique.

Algorithm DPWCFT :

$\Gamma[0 \dots N][0] \leftarrow cost(N);$

for $f = 1$ **to** F **do begin**

$\Gamma[0 \dots N][f] \leftarrow 0;$

for $j = N$ **downto** 0 **do begin**

$C[0 \dots j] \leftarrow cost(j);$

for $i = j$ **downto** 0 **do begin**

$\Gamma[i][f] \leftarrow \max\{C[i] + \Gamma[j][f - 1], \Gamma[i][f]\};$

end

end

end

In the algorithm above, the function $cost(j)$ returns $C(i, j)$ for all i s.t. $0 \leq i \leq j$ as we later describe. Thus, at the end of the algorithm, $\Gamma[0][F]$ will have

$\Gamma(0, F)$ as the final result. The time complexity of the algorithm is $O(N^2F)$ providing that of $cost(j)$ is $O(j)$ as we later show. Its space complexity is $O(NF)$ if we need not only the worst case cost but also the timings, or $O(N)$ otherwise because only $\Gamma[0 \dots N][f]$ and $\Gamma[0 \dots N][f - 1]$ are required for all f .

We now show that $cost(j)$ is evaluated in time $O(j)$. If we know $m_{e(k)}^\gamma(k, j)$ for $\gamma = 0$ and 1 , for example, $m_{e(k)}^0(k', j)$ for all $p(k) \leq k' < k$ is calculated by

$$m_{e(k)}^0(k', j) = \begin{cases} m_{e(k)}^1(k, j) + 1 & b_k \mapsto T \\ m_{e(k)}^0(k, j) & b_k \mapsto N \end{cases}$$

That is, if $c[e(k)]$ is set to 0 somewhere between $b_{p(k)}$ and b_k , and b_k is a taken branch, b_k is mispredicted and $c[e(k)]$ becomes 1 , resulting in the total number of mispredictions being $m_{e(k)}^1(k, j) + 1$. If b_k is a not-taken one, on the other hand, its action is correctly predicted and $c[e(k)]$ remains 0 . Since similar equations are obtained for $\gamma \geq 1$ and $m_e^\gamma(j, j) = 0$ for any e, γ and j by definition, it is easy to calculate $m_e^\gamma(i, j)$ for all γ and i such that $0 \leq i \leq j$ in time $O(j)$. Therefore, it is also easy to have an $O(j)$ algorithm to calculate $m_e(i, j)$ by a simple maximization of $m_e^\gamma(i, j)$, and $C(i, j)$ using the recurrence $C(i - 1, j) = C(i, j) + (m_{e(i)}^\gamma(i - 1, j) - m_{e(i)}^\gamma(i, j))$

derived from Definition 2. The complete definition of $cost(j)$ is given in [16].

4.2 Saturating Branch Sequence

Since N may be a huge number, some hundred millions in the majority of SPEC CPU95 benchmarks, the $O(N^2F)$ algorithm should be impractical. Thus we have to improve *DPWCFT* to make its time complexity $O(NF)$ for, at least, practical programs to be analyzed. Our targets for the complexity reduction are $cost(j)$ and the innermost loop of *DPWCFT* both of which have time complexity $O(j)$ to be improved to $O(1)$.

The key idea for the improvement is that a specific sequence of branches associated with a counter guarantees its saturation independently from its initial value. For example, three consecutive taken branches, namely TTT, referring to a counter always makes its value 3 after their execution. In general, a sequence TT(NT)*T infallibly saturates the counter with 3, while NN(TN)*N does so with 0. Such sequences, namely *saturating branch sequences* (SBS), are obviously detectable by a simple finite automaton to accept the regular expression TT(NT)*T + NN(TN)*N in time $O(N)$.

Since $\Gamma(i, F)$ for all $i > 0$ are needless, the last iteration for $f = F$ can be modified to calculate $\max_j \{C(0, j) + \Gamma(j, F - 1)\}$ as is done in our real implementation.

Although not shown in the algorithm for the sake of simplicity, it is easy to keep the value of j that maximizes $C(i, j) + \Gamma(j, f - 1)$ for all i and f .

The effect of SBS for the improvement is twofold. Suppose b_h and b_t are the head and tail of an SBS whose members refer to the counter $c[e]$ (i.e., $e = e(h) = e(t)$). Since $c[e]$ definitely becomes 0 or 3 after the execution of b_h, \dots, b_t , the number of mispredictions $m_e(i, j)$ for all $i < h$ and $j \geq t$ can be calculated by the following formulae:

$$\begin{aligned} \vec{m}_e(t, j) &= \begin{cases} m_e^0(t, j) & b_t \mapsto \mathbb{N} \\ m_e^3(t, j) & b_t \mapsto \mathbb{T} \end{cases} \\ m_e(i, j) &= m_e(i, t) + \vec{m}_e(t, j) \end{aligned} \quad (3)$$

From equation (3) above, we have the following two recurrences for $C(i, j)$, where $\bar{h}(j)$ and $\bar{t}(j)$ are the indices of the head and tail of the most recent SBS of $e(j)$ preceding b_j , while $\bar{h}(i)$ and $\bar{t}(i)$ mean those of the earliest SBS of $e(i)$ following b_i .

$$\begin{aligned} \forall i < \bar{h}(j) : C(i, j-1) &= C(i, j) - (m_{e(j)}(i, j) - m_{e(j)}(i, j-1)) \\ &= C(i, j) - (\vec{m}_{e(j)}(\bar{t}(j), j) - \vec{m}_{e(j)}(\bar{t}(j), j-1)) \\ &\equiv C(i, j) - \vec{\delta}(j) \end{aligned} \quad (4)$$

$$\begin{aligned} \forall j \geq \bar{t}(i) : C(i-1, j) &= C(i, j) + (m_{e(j)}(i-1, j) - m_{e(j)}(i, j)) \\ &= C(i, j) + (m_{e(i)}(i-1, \bar{t}(i)) - m_{e(i)}(i, \bar{t}(i))) \\ &\equiv C(i, j) + \vec{\delta}(i) \end{aligned} \quad (5)$$

Equation (4) means $C(i, j-1)$ can be calculated for all $i < \bar{h}(j)$ from $C(i, j)$ and $\vec{\delta}(j)$, which is independent from i , rather than calculated iteratively as $cost(j)$ does. It is also derived from (5) that $C(i-1, j)$ can be calculated for all $j \geq \bar{t}(i)$ from $C(i, j)$ and $\vec{\delta}(i)$, which is independent from j , rather than calculated iteratively as the innermost loop of $DPWCFT$ does.

4.3 Improvement of $cost()$

From equation (4), $cost(j-1)$ for the calculation of $C(i, j-1)$ for all $i < j$ is replaced with the following equivalent operations as illustrated in **Fig. 2**. First, let $H(j) = \{h_1, \dots, h_{m_j}\}$ be the set of SBS head indices $h_1 < \dots < h_{m_j}$ that satisfy the following:

$$H(j) = \{h \mid h = \bar{h}(k), k > j, \bar{t}(k) \leq j, j < \forall l < k : [\bar{h}(k) < \bar{h}(l)]\}$$

For example, in the upper half of the figure, $h_1 = \bar{h}(j+3)$, $h_2 = \bar{h}(j+2)$ and $h_3 = \bar{h}(j+1)$, which are left edges of the rectangles representing SBS, are members of $H(j)$, but $\bar{h}(j+4)$ is not because $\bar{h}(j+3) \leq \bar{h}(j+4)$.

Each member of $H(j) = \{h_1, \dots, h_{m_j}\} = \{\bar{h}(k_1), \dots, \bar{h}(k_{m_j})\}$, where $k_l = \min\{k > j \mid \bar{h}(k) = h_l\}$ and $k_0 = \min(\{k > j \mid \bar{h}(k) = 0\} \cup \{N+1\})$, has an associated *differential cost*

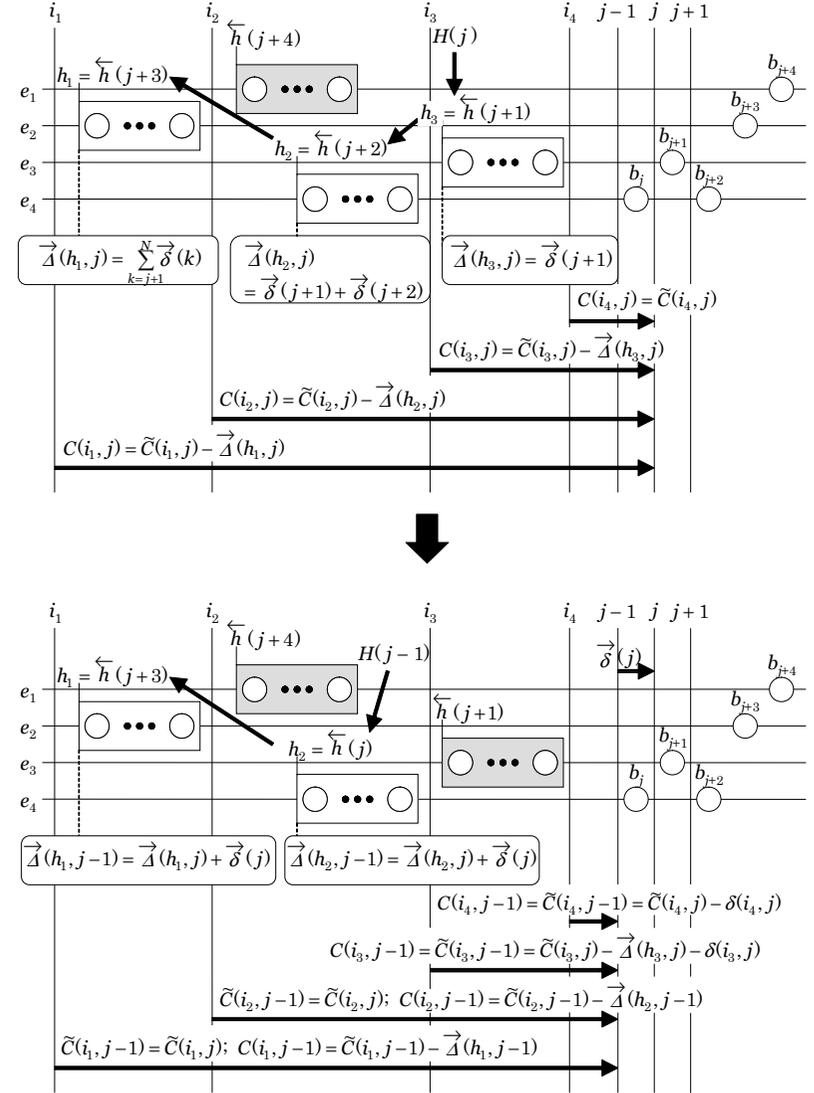


Fig. 2 Calculating $C(i, j-1)$ from $C(i, j)$.

$$\vec{\Delta}(h_l, j) = \sum_{k=j+1}^{k_{l-1}-1} \vec{\delta}(k) \quad (6)$$

Thus, in the upper half of the figure, $\vec{\Delta}(h_l, j)$ for h_1, h_2 and h_3 have the values shown in the figure.

Finally, for each $i \leq j$, we define the *base cost* $\tilde{C}(i, j)$ partially calculated by improved $\text{cost}()$ as follows:

$$\tilde{C}(i, j) = \begin{cases} C(i, j) + \vec{\Delta}(h_1, j) & i < h_1 \\ C(i, j) + \vec{\Delta}(h_l, j) & h_{l-1} \leq i < h_l, \quad 1 < l \leq m_j \\ C(i, j) & h_{m_j} \leq i \end{cases} \quad (7)$$

Note that h_{m_j} is considered as 0 if $H(j) = \emptyset$. Also note that since $\vec{\Delta}(h_1, j) = \sum_{k=j+1}^N \vec{\delta}(k)$ from (6) and $C(i, j) = C(i, N) - \vec{\Delta}(h_1, j)$ for all $i < h_1$ by the iterative applications of (4), we have $\tilde{C}(i, j) = C(i, N)$ for all $i < h_1$. The final remark is that $C(i, j)$ is calculated from $\tilde{C}(i, j)$ and $\vec{\Delta}(h_l, j)$ with h_l appropriate to i , which will be found by a binary search of the data structure (a simple array) for $H(j)$.

Now we show how $H(j-1)$, $\vec{\Delta}(h_l, j-1)$, and $\tilde{C}(i, j-1)$ are calculated from their counterparts of j . We descendingly scan branches b_i from $i = j$ to $i = \bar{h}(j)$. Until we find $i = h_{m_j} \in H(j)$, we calculate $\tilde{C}(i, j-1)$ through the following equation.

$$\begin{aligned} \delta(i, j) &= m_{e(j)}(i, j) - m_{e(j)}(i, j-1) \\ \tilde{C}(i, j-1) &= \tilde{C}(i, j) - \delta(i, j) \\ &= C(i, j) - \delta(i, j) = C(i, j-1) \end{aligned}$$

If we find $i = h_{m_j}$, we *pop* it from $H(j)$ and switch the equation to

$$\begin{aligned} \tilde{C}(i, j-1) &= \tilde{C}(i, j) - \vec{\Delta}(h_{m_j}, j) - \delta(i, j) \\ &= C(i, j) - \delta(i, j) = C(i, j-1) \end{aligned}$$

and do the same each time $i = h_l$ is found. Before we reach $i = \bar{h}(j)$, we must encounter $i = \bar{t}(j)$ at which we have $\vec{\delta}(j) = \delta(\bar{t}(j), j)$ according to its definition in (4). Finally, we reach $i = \bar{h}(j)$, and *push* it onto the remaining members of $H(j)$ to form $H(j-1)$ if $\bar{h}(j) \notin H(j)$. Then, for all $h_l \in H(j-1)$, $\vec{\delta}(j)$ is added to $\vec{\Delta}(h_l, j)$ to have $\vec{\Delta}(h_l, j-1)$ and to keep the following

If we do not have $k > j$ such that $\bar{h}(k) = 0$, otherwise N in this discussion should be replaced with $k_0 - 1$ where $k_0 = \min\{k > j \mid \bar{h}(k) = 0\}$
 $\bar{h}(j)$ is not pushed if it is 0; i.e., b_j does not have the preceding SBS.
 If $h(j) \notin H(j)$, $\vec{\Delta}(\bar{h}(j), j)$ is considered as that for the element popped last, or zero if nothing has been popped.

invariant for all $i < \bar{h}(j)$.

$$\begin{aligned} \tilde{C}(i, j-1) &= \tilde{C}(i, j) + \vec{\Delta}(h_l, j) \\ &= C(i, j-1) + (\vec{\delta}(j) + \vec{\Delta}(h_l, j)) \\ &= C(i, j-1) + \vec{\Delta}(h_l, j-1) \end{aligned} \quad (*)$$

Note that equation (*) is derived from (4). The important point of the equation above is that we can omit the calculation for $\tilde{C}(i, j-1)$ for all $i < \bar{h}(j)$, and thus have a procedure of $O((j - \bar{h}(j)) + |H(j-1)|)$ to perform operations equivalent to $\text{cost}(j-1)$, namely $\widetilde{\text{cost}}(j-1)$, paying a cost of $\log |H(j-1)|$ for each reference of $C(i, j-1)$ for the binary search of $H(j-1)$.

The result of $\widetilde{\text{cost}}(j-1)$ applied to the upper half of Fig. 2 is shown in the lower half. The complete definition of $\widetilde{\text{cost}}()$ and the proof of its correctness are given in 16).

4.4 Improvement of DPWCFT

The next target is the innermost loop of *DPWCFT* in which we evaluate the following to obtain $\Gamma_i(i, f) = \Gamma(i, f)$ at the end of the outer j loop.

$$\begin{aligned} \Gamma_j(i, f) &= \max(C(i, j) + \Gamma(j, f-1), \Gamma_{j+1}(i, f)) \\ &= \max_{j \leq k \leq N} \{C(i, k) + \Gamma(k, f-1)\} \end{aligned} \quad (9)$$

Combining equation (9) above and (5) from Section 4.2, we have the following recurrence to calculate $\Gamma_j(i-1, f)$ for any j s.t. $j \geq \bar{t}(i)$:

$$\begin{aligned} \Gamma_j(i-1, f) &= \max_{j \leq k \leq N} \{C(i-1, k) + \Gamma(k, f-1)\} \\ &= \max_{j \leq k \leq N} \{C(i, k) + \vec{\delta}(i) + \Gamma(k, f-1)\} \\ &= \vec{\delta}(i) + \max_{j \leq k \leq N} \{C(i, k) + \Gamma(k, f-1)\} \\ &= C(i-1, j) - C(i, j) + \Gamma_j(i, f) \end{aligned} \quad (10)$$

This means that we may not calculate $\Gamma_j(i-1, f)$ for all $j \geq \bar{t}(i)$ but do so only for $j < \bar{t}(i)$ with the initial value derived from $\Gamma_j(i, f)$ if we already know it.

This optimization requires two modifications of the loop structure of *DPWCFT*. First, the set of branch indices $A(j) = \{i \mid i \leq j < \bar{t}(i)\}$, whose elements are the targets of $\Gamma_j(i-1, f)$ calculation, is *inconsecutive*; i.e., there may be $i < i' \leq j$ such that $i \in A(j)$ but $i' \notin A(j)$. Thus, as shown in **Fig. 3**, we need to maintain $A(j)$ as the union of $A(e, j) = \{i \mid e(i) = e, i \leq j < \bar{t}(i)\}$ (double-headed arrows in the figure) for all e . This maintenance, however, is easy because members of $A(e, j)$ are easily found by following the *links* to connect branches sharing a counter; i.e., b_i to $b_{p(i)}$.

Second, when we find b_j being the tail of an SBS, we need not only add new

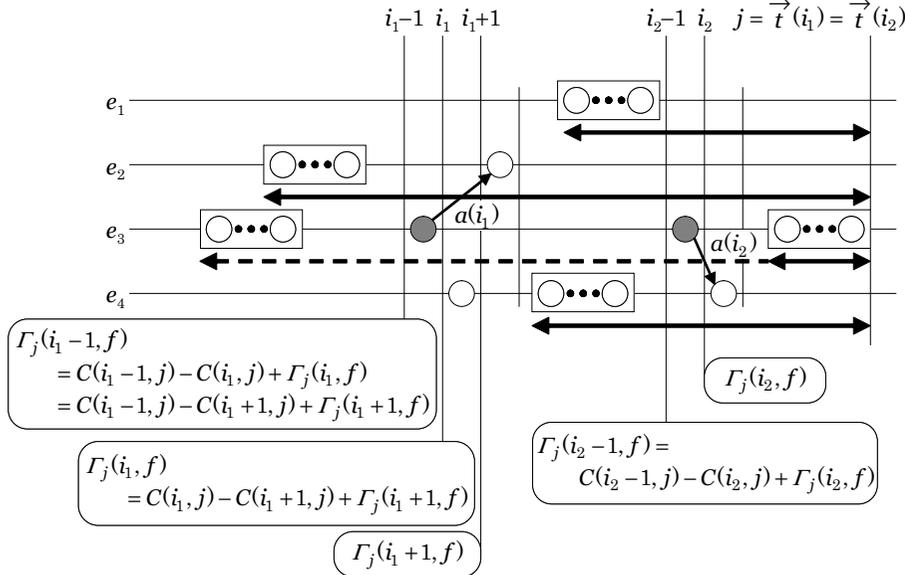


Fig. 3 Optimal calculation of $\Gamma_j(i, f)$.

members $\{i \mid \vec{t}(i) = j\}$ (dashed arrow in Fig. 3) to $A(j)$ to have $A(j-1)$, but also have to initialize $\Gamma_j(i-1, f)$ for them by equation (10). This is easy if $i+1 \in A(j)$ because we have already had $\Gamma_j(i, f)$ (as $i = i_2$ in the figure), but we have to pay some attention for the case of $i = i_1$ in the figure where $i_1 + 1 \notin A(j)$. This problem is solved by iteratively applying the recurrence (10), as shown in the figure, because $\vec{t}(i+1) > j$ if $i+1 \notin A(j)$. In general, we can find $a(i) = \min\{k \mid i < k, \vec{t}(i) \leq \vec{t}(k)\}$ by scanning all branches once and in advance, to calculate $\Gamma_j(i-1, f)$ through the following:

$$\Gamma_j(i-1, f) = C(i-1, j) - C(a(i)-1, j) + \Gamma_j(a(i)-1, f) \quad (11)$$

Now we have an optimized version of $DPWCFT$, namely $OptWCFT$, which is logically equivalent to $DPWCFT$ and whose outline is illustrated as follows:

Algorithm OptWCFT :

$\Gamma[0 \dots N][0] \leftarrow cost(N)$; find $a(i)$ for all i ;

for $f = 1$ **to** F **do begin**

$\Gamma[0 \dots N][f] \leftarrow 0$;

```

 $\tilde{C}[0 \dots N] \leftarrow cost(N)$ ;
 $H \leftarrow \emptyset$ ;  $A \leftarrow A(N)$ ;
for  $j = N$  downto 1 do begin
  for  $\forall i \in A$  do
     $\Gamma[i-1][f] \leftarrow \max(C(i-1, j) + \Gamma[i-1][f-1], \Gamma[i-1][f])$ ;
   $A \leftarrow A - \{j\}$ ;
  if  $b_j$  is an SBS tail then begin
     $A' \leftarrow \{i \mid \vec{t}(i) = j\}$ ;
    for  $\forall i \in A'$  do
       $\Gamma[i-1][f] \leftarrow C(i-1, j) - C(a(i)-1, j) + \Gamma[a(i)-1][f]$ ;
       $A \leftarrow A \cup A'$ ;
    end
     $(H, \tilde{C}[]) \leftarrow \widetilde{cost}(j-1)$ ;
  end
end

```

Note that the initial value of $A = A(N)$ has the indices of branches which follow the last SBS of each counter. Also note that $C(i, j)$ is calculated by the equation (??) from $\tilde{C}[i] = \tilde{C}(i, j)$ and $\tilde{\Delta}(h, j)$ associated with each element of $H = H(j)$ with a binary search of it. The complete definition of $OptWCFT$ and the proof of its correctness are given in 16).

4.5 Complexity of $OptWCFT$

To evaluate the time complexity of $OptWCFT$, we need a few definitions. Let B be the number of counters referred to in the execution to be analyzed. Obviously, $B \leq |P|$ and is bounded by the *static* count of branches in the program even if P is infinitely large. Let $\sigma(j)$ be the set of branch indices $\{i \mid \vec{h}(i) = j\}$ preceding an SBS head b_j , and L be the average of $|\sigma(j)|$. Finally, let λ be the average number of branches in an SBS.

First, the time complexity of $\widetilde{cost}(j-1)$ is $O((j - \vec{t}(j)) + |H(j-1)|)$, the average of $j - \vec{t}(j)$ is $B(L + \lambda)/2$, and $|H(j-1)| \leq B$. Thus, its average time complexity is evaluated as $O(B(L + \lambda))$. Second, since the average of $|A(j)|$ is again $B(L + \lambda)/2$ and it is the number of iterations of the innermost loop of $OptWCFT$ (the first one) in which we need $\log |H(j)| \leq \log B$ operations for a binary search to calculate $C(i, j)$, the average time complexity of the loop is evaluated as $O(B \log B \cdot (L + \lambda))$. Thus, the total time complexity of $OptWCFT$ is $O(NB \log B \cdot (L + \lambda)F)$.

Now we estimate L and λ , assuming that each branch is randomly taken or not taken. This assumption is not the worst for our algorithm, but is sufficiently pessimistic because it means the prediction accuracy is 50%. With this assumption, the probability of a branch being the head of SBS (TT(NT)*T

or $NN(TN)^*N$, namely p_S , is $1/3$ because $p_S = 2(1/2^3 + 1/2^5 + 1/2^7 + \dots)$. Thus, it is obvious that $L = 1/p_S = 3$. The expected value of λ is calculated from the fact that the probability of the length of an SBS being $2k+1$ is $3(1/4)^k$ and the formula $\sum_{k=1}^{\infty} kx^{k-1} = 1/(1-x)^2$. That is, $\lambda = \sum_{k=1}^{\infty} 3(2k+1)/4^k = 11/3$. Thus, $L + \lambda$ is a small constant of $20/3$ even with the random branch assumption, and is smaller than the number in practical program executions as measured by our evaluation, as we later discuss.

We can then consider that B is a constant independent from and much smaller than N . Thus, we can conclude that the time complexity of *OptWCFT* is $O(NF)$ in a practical sense.

As for the space complexity, *OptWCFT* requires the following data structures. First, each of N branches, b_i , may be represented by a constant number of elements for $e(i)$, $p(i)$, $a(i)$, $\bar{h}(i)$, $\bar{t}(i)$, $\bar{h}(i)$ and $\bar{h}(i)$ and a Boolean flag to indicate $b_i \mapsto T$ or $b_i \mapsto N$. The other structures, whose sizes are proportional to N , are $\Gamma[0 \dots N][0 \dots F]$ to hold $\Gamma(i, f)$ for each i and f , and $\tilde{C}[0 \dots N]$ to hold $\tilde{C}(i, j)$ for each i and a particular j . We also need the following $O(|P|)$ size structures: an array of $4|P|$ elements to hold $m_e^\gamma(i, j)$ in the calculation of $cost()$ and $\widetilde{cost}()$; a stack H to hold $H(j)$ and $\tilde{\Delta}(h, j)$ associated to each element $H(j)$; and an array to have $\alpha(i, e) = \max\{k | b_k \in A(i), e(k) = e\}$ for each e to represent $A(i)$ together with the link of b_j to $b_{p(j)}$.

The space complexity of *OptWCFT* is thus $O(NF + |P|)$, or $O(NF)$ because $|P|$ is a constant that is usually much smaller than N , if we need the worst case timings as discussed in Section 4.1. Otherwise, if only the worst case delay is required, so only $\Gamma[0 \dots N][f]$ and $\Gamma[0 \dots N][f-1]$ are necessary, the complexity is $O(N)$, also as discussed in Section 4.1

5. Experiments

We implemented two algorithms *DPWCFT* and *OptWCFT*, and evaluated their performance with SPEC CPU95 benchmarks and a bimodal predictor of $|P| = 2,048$. The benchmarks, except for *jpeg*, are rarely executed in real-time systems, but their widely varying characteristics should be useful for examining how wide an application range our algorithms are effective for. The analysis programs are written in C, compiled by `gcc 3.2.2` with the `-O2` option, and run on a Linux PC of 2.0 GHz Opteron-246 with 2 GB memory.

As the input of our analyzer, we obtained the trace of conditional branches executed in each benchmark with “train” data set using an instruction simulator based on `sim-fast` of the SimpleScalar 3.0 tool-set¹⁾ for the PISA instruction set. Statistics regarding the benchmarks are summarized in **Table 1**, from which we confirmed that L and λ are smaller than the estimation in

Table 1 Statistics of benchmarks.

benchmark	$N(\times 10^3)$	B	L	λ	benchmark	$N(\times 10^3)$	B	L	λ
099.go	62,098	1,902	1.65	3.36	102.swim	28,859	790	1.06	3.05
124.m88ksim	6,496	936	1.11	3.11	103.su2cor	792,679	1,032	1.08	3.03
126.gcc	198,388	2,048	1.28	3.16	104.hydro2d	739,074	1,117	1.01	3.00
129.compress	3,889	427	1.39	3.25	107.mgrid	205,374	812	1.11	3.10
130.li	24,233	587	1.26	3.16	110.applu	17,842	912	1.80	3.60
132.jpeg	97,793	1,103	1.18	3.10	125.turb3d	734,122	1,075	1.26	3.20
134.perl	328,327	1,327	1.16	3.12	141.apsi	100,362	1,169	1.11	3.07
147.vortex	299,712	1,979	1.04	3.03	145.fpppp	3,385	806	1.16	3.09
101.tomcatv	361,766	835	1.04	3.03	146.wave5	192,736	1,132	1.02	3.01

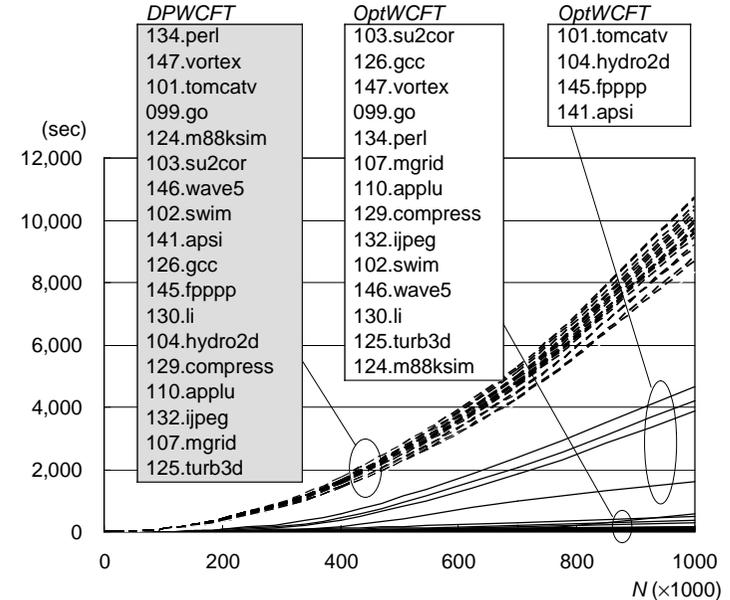


Fig. 4 Execution time of *DPWCFT* and *OptWCFT*.

Section 4.5.

Figure 4 shows the execution times of two algorithms for each benchmark. Up to one million leading branches were analyzed for each benchmark with the parameter $F = 2$. In the graph, the results for *DPWCFT* are represented by dashed lines and in the shaded box the benchmarks are listed in descending order by the execution times for $N = 10^6$. The graph clearly shows that *DPWCFT* requires a time of $O(N^2)$, which is almost independent of each benchmark's nature, and is distributed in a range between 8,310 to 10,737 seconds (or about 2.5 hours) when $N = 10^6$.

On the other hand, the results for *OptWCFT* (solid lines and unshaded boxes) strongly depend on the benchmark characteristics. From the graph, we can observe that benchmarks are categorized into two groups, one with tomcatv, hydro2d, fpppp and apsi, and the other with remaining benchmarks. In the second group, *OptWCFT* greatly outperforms *DPWCFT*, as expected, by up to 173-fold and 80-fold on average when $N = 10^6$. As anticipated, we also found that *OptWCFT* takes $O(N)$ time with about 5,000 branches per second on average.

The performance curves of the first group, however, are quite different and look non-linear, especially for the slowest three benchmarks, giving counter-examples to the complexity analysis discussed in Section 4.5. However, we consider their curves to be linear but with steep slopes after N exceeds about 400 thousand. This belief is supported by the fact that these program executions have branch sequences hardly saturating the counter they commonly refer to, resulting in a very long SBS of more than 150 branches. Such a long SBS does not affect the time complexity because it is affected by the average length, which is small even in these benchmarks; however, it strongly degrades real computation time by decreasing access locality. That is, occasional long SBS occurrences force $\widehat{cost}()$ and the innermost loop to scan widely distributed and/or a large number of branches causing frequent cache misses and even TLB misses.

The worst case misprediction counts shown in **Table 2** also depend on characteristics of the benchmarks as one would expect. Notably, the number of mispredictions caused by two flushes, d in the table, cannot be estimated from, for example, the proportion of B . That is, the increment of mispredictions per counter, d/B , widely varies from 1.7 of wave5 to 21.0 of fpppp. Another insight

Table 2 Misprediction count ($\times 10^3$, $N = 10^6$).

	$F=2$	$F=0$	d^\dagger
go	154.2	146.5	7.8
m88ksim	55.1	52.7	2.5
gcc	114.6	107.1	7.6
compress	171.0	165.6	5.4
li	113.0	111.3	1.6
jpeg	95.8	93.3	2.6
perl	46.1	42.8	3.3
vortex	48.3	40.4	7.9
tomcatv	66.6	52.6	14.0
swim	45.0	42.8	2.3
su2cor	191.5	184.4	7.1
hydro2d	59.6	44.6	15.0
mgrid	45.8	42.5	3.3
applu	236.7	234.1	2.6
turb3d	7.2	5.3	1.9
apsi	102.2	89.9	12.3
fpppp	102.5	85.6	16.9
wave5	2.9	1.1	1.9

$\dagger d$ is the difference of mispredictions between $F = 2$ and $F = 0$.

gained from the table is that the four benchmarks having highest d —fpppp, hydro2d, tomcatv and apsi—are those which *OptWCFT* takes a long time to analyze. This is not an accident but is explained again by the long SBS in their execution. A long SBS has a long sequence of $(TN)^*$ that we would expect to cause $(5/8)l$ mispredictions on *average*, where l is the length of the sequence, because $l/2$ misses result from three out of four possible counter values at its beginning. The sequence, however, suffers l mispredictions—i.e., fails always—with the *worst case* counter value. These insights strongly support the importance of accurate worst case analysis.

The large values of d also support our claim that the interruption delay caused by the flush of the predictor is as significant as that of caches. First, d being larger than 10,000 for four benchmarks means that the prediction accuracy is degraded by more than 1%, which is considered significant by branch predictor architects. For example, two recent papers on predictors 7) and 9) discuss new prediction mechanisms which improve the accuracy by *less* than 1%. This does not mean their proposals are insignificant but proves a great impact on the overall performance can result from a small accuracy improvement. In fact, the first paper 7) reports that a 0.5% accuracy improvement gives a 4.5% performance gain, while the second paper 9) claims

The time needed for the instruction simulation to obtain the trace of one-million branches is excluded from the execution time because it is negligibly small, less than 3 seconds on average.

a 0.6% accuracy improvement results in a 17% performance gain. In general, it is widely believed in this community that a 1% accuracy improvement/degradation results in a 10% performance gain/loss.

Further support comes from comparing the amount of delay caused by flushes of a predictor and a cache. Let us assume that two flushes of the predictor increase its misses by 15,000 as in the case of hydro2d, and each miss costs 20 cycles as in the case for modern microprocessors with deep pipelines such as the Pentium 4. This means the total delay is 300,000 cycles. As for the cache, let us assume we have 32KB level-1 instruction and data caches with 32B block size. This means that a flush causes each cache to lose up to 1,024 useful blocks, so two flushes cause two caches to suffer up to 4,096 additional misses in total. In the worst case, these misses also cause misses in a level-2 (and higher levels if so equipped) cache with a large delay for physical memory access, say 100 cycle per miss, resulting a total delay of 409,600 cycles. These two delays, 300,000 and 409,600 cycles, are clearly comparable. Moreover, since caches may not be fully filled with useful blocks (roughly half or less of the total blocks were useful in the experiment reported in 15)), the delay caused by the prediction misses may be larger than that due to the cache misses.

6. Application to Other Predictors

In this section, we discuss the applicability of our scheme to other predictors for branch direction. For example, *gselect*¹⁹⁾ and *gshare*¹³⁾ predictors, whose 2-bit counter table is indexed by a function of the *global branch history*, will make the story complicated because the history is also *flushed*. This uncertainty problem of counter indices will be directly solved by searching for the worst case in 2^g *multiple worlds* built with a g bit global history as follows.

Let $e(i, \eta)$ be a function to obtain the counter $c[e(i, \eta)]$ referred to by b_i with a global history η . Let us assume that a flush just after b_i makes the global history some specific value x . With this assumption we can easily calculate the sequence of global histories $x = \eta_{i+1}^x, \eta_{i+2}^x, \dots, \eta_N^x$ for branches b_{i+1}, \dots, b_N . Note that η_j^x depends on x if $j \leq i + g$, but is independent from it if $j > i + g$ because η_j^x is determined by the action (taken or not taken) of b_{j-g}, \dots, b_{j-1} and thus by scanning all the branches once in advance.

Now we have a recurrences such as

$$m_{e(k, \eta_k^x)}^0(k', j) = \begin{cases} m_{e(k, \eta_k^x)}^1(k, j) + 1 & b_k \mapsto \text{T} \\ m_{e(k, \eta_k^x)}^0(k, j) & b_k \mapsto \text{N} \end{cases}$$

for all k' such that $p(k) \leq k' < k$, where $p(k) = \max\{l | e(k, \eta_k^x) = e(l, \eta_l^x), l < k\}$ as is done in Section 4.1. Therefore, we have $m_e(i, j, x) =$

$\max_{0 \leq \gamma \leq 3} \{m_e^\gamma(i, j)\}$ for all e and thus $C(i, j, x) = \sum_e m_e(i, j, x)$ for the initial history x following the Definition 2. Finally, we have the flush cost $C(i, j) = \max_{0 \leq x < 2^g} \{C(i, j, x)\}$. Note that the recurrences

$$\begin{aligned} x_i &= 2x_{i-1} + \begin{cases} 1 & b_i \mapsto \text{T} \\ 0 & b_i \mapsto \text{N} \end{cases} \pmod{2^g} \\ C(i-1, j, x_{i-1}) &= C(i, j, x_i) + \\ &\quad m_{e(i, x_{i-1})}(i-1, j, x_{i-1}) - m_{e(i, x_{i-1})}(i, j, x_i) \end{aligned}$$

are also derived from the definition and the history maintenance mechanism by a shift register. Therefore, a version of *cost(j)* with a global history can be designed to compute $C(i, j)$ for all $i \leq j$ in time $O(2^g N)$, or $O(N)$ if we can assume 2^g is a constant.

Alternatively, if the 2^g factor is too large, a more efficient solution will be obtained by assuming that a flush lets counters have the worst case pattern *after* g branches following it are executed. That is, since $\eta_i = \eta_i^x$ for all $i > g$ can be determined in advance independently of the initial history x , $C(i, j, \eta_i)$ for all i and j such that $g < i \leq j$ may be calculated without exploring the 2^g multiple worlds. Then a conservative definition,

$$C(i, j) = \begin{cases} j - i & j < i + g \\ C(i + g, j, \eta_{i+g}) + g & j \geq i + g \end{cases}$$

which assumes up to g leading branches are always mispredicted, gives us another version of *cost()* which should be as efficient as that without a global history. Although this simplification is a little bit pessimistic because the worst case counter pattern deriving $C(i + g, j, \eta_{i+g})$ could be infeasible with any initial global history, the higher efficiency will compensate for the pessimism.

Note that the concept of SBS is still useful with both the direct and simplified methods shown above, but we have to modify the base formula (4) and the definitions of $\vec{h}(i)$ and $\vec{t}(i)$. That is, the qualifier $\forall i < \vec{h}(j)$ in (4) should be rewritten as $\forall i < \vec{h}(j) - g$, while $\vec{h}(i)$ and $\vec{t}(i)$ should be the head and tail of the SBS that is the earliest one following b_{i+g} rather than b_i .

Finally, a predictor with multiple prediction mechanisms, such as *combined*¹³⁾, will be also targetable for our scheme if it consists of multiple tables of 2-bit counters optionally with a global history. For a combined predictor with a bimodal and *gselect/gshare* predictors whose predictions are selected by a *meta*-predictor, for example, $C(i, j)$ may be computed by exploring $2^{2+2+2} = 64$ cases to calculate $m_e(i, j)$ if we adopt the simplified method for the *gselect/gshare* predictor. That is, with three tables and thus three counter values, namely γ_b of bimodal, γ_g of *gselect/gshare*, and γ_m of *meta*-predictor, we have to keep track of the number of misses for each of 64 cases of $0 \leq \gamma_b \leq 3$,

$0 \leq \gamma_g \leq 3$ and $0 \leq \gamma_m \leq 3$. Additionally, since each predictor should have its own SBS, our optimization is applicable using the *furthest* one of the three; i.e., the earliest for $\widehat{cost}()$ and the latest for *OptWCFT*'s innermost loop.

7. Conclusion and Discussion

We proposed two algorithms for the WCFT problem of bimodal branch predictors. First we formulate the problem to confirm it is solvable by a dynamic programming technique and to provide an $O(N^2F)$ algorithm *DPWCFT*. The concept of SBS is then introduced, by which we improve the time complexity to $O(NF)$. The improved algorithm *OptWCFT* greatly outperforms *DPWCFT* in the analysis of SPEC CPU95 benchmarks, thus showing the significance of our optimization.

There are many directions to follow to extend of our contribution in the future. The most important direction is to apply our method to other predictors as discussed in Section 6. Another direction is to incorporate other hardware components such as caches, TLBs, and branch target buffers into our analysis scheme to approach the ultimate goal, an accurate estimation of interruption/preemption delay. The incorporation will be straightforward if a good estimation is given for the total cost caused by misses of these components whose worst case behaviors can be analyzed by algorithms similar to ours or Miyamoto's¹⁴. We must take care in the incorporation, though, since each component has its own event patterns to disconnect causality—e.g., an SBS for a bimodal predictor—and so optimization to exploit them will become more difficult.

We have already partially explored a slightly different direction towards the ultimate goal as reported in 10) and 17). In these papers, we proposed an efficient method to estimate the worst case delay with a cycle accurate simulator for the case of $F = 1$. Although this simulation-based analyzer is very accurate with a realistic hardware model and is reasonably fast (2 to 6 minutes per one million executed instructions with its $O(N \log N)$ time complexity), we still need an unacceptably long $O(N^2F)$ time to apply this method to the cases of $F > 1$. Thus, efficient trace-based methods for branch predictors and caches should be combined with the simulation-based method so that, for example, we can focus on interruption points with large miss counts to reduce the complexity.

Another direction is to analyze *partial flush* cases in which one of a number of predefined sets of counters is flushed by a preemption. Since this type of flush does not disconnect the causality, it is an open problem to design efficient algorithms that should be much faster than a straightforward $O(N^F)$ one.

Acknowledgments This research is partly supported by Grant-in-Aid of Scientific Research #17300015 of MEXT Japan, and the 21st Century COE Program entitled “Intelligent Human Sensing” of MEXT Japan.

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) Bate, I. and Reutemann, R.: Worst-Case Execution Time Analysis for Dynamic Branch Predictors, *ECRTS'04*, pp.215–222 (2004).
- 3) Bate, I. and Reutemann, R.: Efficient Integration of Bimodal Branch Prediction and Pipeline Analysis, *RTCSA 2005*, pp.39–44 (2005).
- 4) Colin, A. and Puaut, I.: Worst Case Execution Time Analysis for a Processor with Branch Prediction, *Real-Time Systems*, Vol.18, No.2/3, pp.249–274 (2000).
- 5) Engblom, J. and Ermedahl, A.: Pipeline Timing Analysis Using a Trace-Driven Simulator, *RTCSA'99*, pp.88–95 (1999).
- 6) Healy, C.A., Arnold, R.D., Mueller, F., Whalley, D.B. and Harmon, M.G.: Bounding Pipeline and Instruction Cache Performance, *IEEE Trans. Computers*, Vol.49, No.1, pp.53–70 (1999).
- 7) Ishii, Y. and Hiraki, K.: Path Trace Branch Prediction, *IPSJ Trans. Advanced Computing Systems*, Vol.47, No.SIG 3(ACS13), pp.58–72 (2006). In Japanese.
- 8) Kim, S.-K., Min, S.L. and Ha, R.: Efficient Worst Case Timing Analysis of Data Caching, *RTAS'96*, pp.230–240 (1996).
- 9) Kise, K., Katagiri, T., Honda, H. and Yuba, T.: The Bimode-Plus Branch Predictor, *IPSJ Trans. Advanced Computing Systems*, Vol.46, No.SIG 7(ACS10), pp.85–102 (2005). In Japanese.
- 10) Konishi, M., Nakada, T., Tsumura, T. and Nakashima, H.: Measuring Worst-Case Performance of Microprocessor by Interruption with Omitting Redundant Execution, *IPSJ Trans. Advanced Computing Systems*, Vol.47, No.SIG12(ACS15), pp.159–170 (2006). In Japanese.
- 11) 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).
- 12) Li, X., Mitra, T. and Roychoudhury, A.: Accurate Timing Analysis by Modeling Caches, Speculation and their Interaction, *DAC 2003*, pp.466–471 (2003).
- 13) McFarling, S.: Combining Branch Predictors, Technical Report WRL TN-36, DEC (1993).
- 14) Miyamoto, H., Iiyama, S., Tomiyama, H., Takada, H. and Nakashima, H.: A Search Algorithm of Worst-Case Cache Flush Timings Using Dynamic Programming, *IPSJ Trans. Advanced Computing Systems*, Vol.46, No.SIG 16 (ACS12), pp.85–94 (2005). In Japanese.
- 15) Miyamoto, H., Iiyama, S., Tomiyama, H., Takada, H. and Nakashima, H.: An

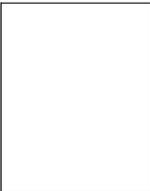
- Efficient Search Algorithm of Worst-Case Cache Flush Timings, *RTCSA 2005*, pp.45–52 (2005).
- 16) Nakashima, H.: WCFT Algorithms for Branch Predictor and Proofs of Their Correctness, Technical report, Kyoto University (2006).
<http://www.para.media.kyoto-u.ac.jp/TR/bpred-alg.pdf>.
 - 17) Nakashima, H., Konishi, M. and Nakada, T.: An Accurate and Efficient Simulation-Based Analysis for Worst Case Interruption Delay, *CASES 2006*, pp. 2–12 (2006).
 - 18) Negi, H.S., Mitra, T. and Roychoudhury, A.: Accurate Estimation of Cache-Related Preemption Delay, *CODES+ISSS 2003*, pp.201–206 (2003).
 - 19) Pan, S.-T., So, K. and Rahmeh, J.T.: Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation, *ASPLOS-V*, pp.76–84 (1992).
 - 20) 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).
 - 21) Smith, J.E.: A Study of Branch Prediction Strategies, *ISCA '81*, pp.135–148 (1981).
 - 22) Tan, Y. and Mooney, V.: Integrated Intra- and Inter-Task Cache Analysis for Preemptive Multi-tasking Real-Time Systems, *SCOPE 2004*, LNCS 3199, pp. 182–199 (2004).
 - 23) White, R., Mueller, F., Healy, C., Whalley, D.B. and Harmon, M.G.: Timing Analysis for Data Caches and Set-Associative Caches, *RTAS'97*, pp.192–202 (1997).

(Received ??? ??, ????)

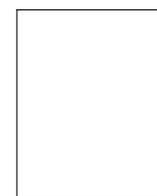
(Accepted ??? ??, ????)



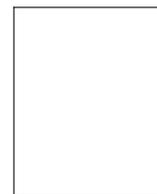
Masahiro Konishi received his M.E. degree from Toyohashi University of Technology in 2006 and joined PFU Ltd. that year. As a student, he engaged in research work on microprocessor simulators.



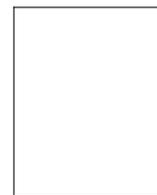
Takashi Nakada received his M.E. degree in 2004 and his Ph.D. in 2007 from Toyohashi University of Technology. He joined Nara Institute of Science and Technology in 2007 as an assistant professor. His current research interests are computer architecture and related simulation technologies. He is a member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



Tomoaki Tsumura received his M.E. and Ph.D. degree from Kyoto University in 1998 and 2004 respectively. After graduating from the Ph.D. candidate course of the Graduate School of Informatics, Kyoto University in 2001, he joined the university as a research associate. He joined Toyohashi University of Technology in 2004, and then joined Nagoya Institute of Technology as an associate professor in 2006. His current research interests are computer architecture, applications of parallel processing, and brain-type information processing. He is a member of IPSJ, JNNS, IEICE and ACM.



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 Board Member of IPSJ, and a member of IEEE-CS, ACM, ALP and TUG.



Hiroaki Takada has been a professor at the Graduate School of Information Science, Nagoya University since 2003, after his academic career as a research associate at the University of Tokyo and as an associate professor at Toyohashi University of Technology. He received his Ph.D. from the University of Tokyo in 1996. He has pursued his research work on embedded systems and has led ITRON and TOPPERS projects to establish the μ ITRON 4.0 specification and to develop an open-source implementation of the ITRON operating system. He received the Sakai award in 2000. He is a representative member of IPSJ as the Chair of SIGEMB, and a member of IEICE, JSSST, ACM and IEEE.