August 1990

UILU-ENG-90-2236
CRHC-90-6

Center for Reliable and High-Performance Computing

COMPILER-ASSISTED SIGNATURE MONITORING

Nancy J. Warter
Wen-mei W. Hwu

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-90-2236 (CRHC-90-6)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) 1101 W. Springfield Ave. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Joint Services Electronics Program 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER N00014-84-C-0149
8c. ADDRESS (City, State, and ZIP Code) 800 N. Quincy St. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
COMPILER-ASSISTED SIGNATURE MONITORING

12. PERSONAL AUTHOR(S)
Warter, Nancy J. and Hwu, Wen-mei W.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	1990 August 9	41

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	performance, memory, signature monitoring, compiler-assisted arc

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

A methodology for applying optimizing compiler techniques to signature monitoring in order to reduce performance overhead and simplify monitor hardware is introduced. We present models for the monitor architecture and the signature placement. The monitor architecture model is designed to keep both the hardware and integration complexities low. Our signature model is designed to insert reference signatures in order to satisfy a bound on the error detection latency. Justifying signatures are inserted on program arcs using an $O(N^2)$ algorithm which is significantly better than previous exponential node insertion algorithms. We use optimizing compiler techniques to customize the signature placement for various target processors and to minimize the performance overhead due to justifying signatures.

continued

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

Experiments were performed to study the performance and memory overheads of our compiler-assisted arc insertion signature monitoring method for a variety of architectures with different branch handling schemes. Using run-time information for processors with delayed branching or branch target buffers improves the performance overhead by approximately 50. However, processors that always fetch the instruction following a branch and squash it if the branch is taken (e.g., the MC68000) are able to hide some of the performance overhead and therefore the run-time information only slightly improves the performance overhead. Using the MC68000 as the target processor, the performance and memory overheads for latencies between 10 and 200 instruction cycles, range from 16 to 4 and from 17 to 11 respectively. After 200 cycles, the overheads remain relatively constant. In general, there is an inverse exponential relationship between the performance and memory overheads and the error detection latency.

Compiler-Assisted Signature Monitoring

Nancy J. Warter *Wen-mei W. Hwu*

August 8, 1990

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
1101 W. Springfield Ave.
University of Illinois at Urbana-Champaign
Urbana, IL 61801

Abstract

A methodology for applying optimizing compiler techniques to signature monitoring in order to reduce performance overhead and simplify monitor hardware is introduced. We present models for the monitor architecture and the signature placement. The monitor architecture model is designed to keep both the hardware and integration complexities low. [1] Our signature model is designed to insert reference signatures in order to satisfy a bound on the error detection latency. Justifying signatures are inserted on program arcs using an $O(N^2)$ algorithm which is significantly better than previous exponential node insertion algorithms. We use optimizing compiler techniques to customize the signature placement for various target processors and to minimize the performance overhead due to justifying signatures.

Experiments were performed to study the performance and memory overheads of our compiler-assisted arc insertion signature monitoring method for a variety of architectures with different branch handling schemes. Using run-time information for processors with delayed branching or branch target buffers improves the performance overhead by approximately 50%. However, processors that always fetch the instruction following a branch and squash it if the branch is taken (e.g., the MC68000) are able to hide some of the performance overhead and therefore the run-time information only slightly improves the performance overhead. Using the MC68000 as the target processor, the performance and memory overheads for latencies between 10 and 200 instruction cycles, range from 16% to 4% and from 17% to 11% respectively. After 200 cycles, the overheads remain relatively constant. In general, there is an inverse exponential relationship between the performance and memory overheads and the error detection latency.

[1] Preliminary research for this paper was presented at FTCS-20[23].

1 Introduction

An efficient concurrent error detection scheme should have good error coverage, be easy to implement, not significantly degrade the target system performance, and have reasonable error detection latency. For embedded concurrent error detection schemes, it is particularly important to keep the implementation complexity low. Otherwise, the additional hardware may actually lower the system reliability. To keep the implementation complexity low, the hardware should be simple and the integration should not require major modifications to the basic system architecture.

In recent years, signature monitoring has become an attractive embedded concurrent error detection scheme because it can detect approximately 99% of the control flow errors [11, 17, 25] using a simple watchdog monitor[2] [15, 12, 16, 20]. In signature monitoring, the compiler encodes the program control flow information into signatures. At run-time, the watchdog monitor uses these signatures to detect instruction bit and sequence errors [21]. Sequence errors correspond to failures that result in incorrect program flow.

In most signature monitoring schemes, signatures are inserted directly into the program code [14, 18, 25]. Adding these signatures degrades the target system performance and increases the program memory requirements. In order to reduce these performance and memory overheads, previous schemes have added hardware assists to the watchdog monitor [15, 19, 25].

In this paper, we present a signature monitoring method which uses optimizing compiler techniques instead of hardware assists to reduce the performance overhead.[3] The optimizing compiler is customized to the target processor so that other than a simple interface, the monitor architecture is target processor independent. Furthermore, signatures are placed such that they guarantee a

[2]Experiments performed by Gunneflo et al. indicate that approximately 78% of the measured errors were control flow errors [7].

[3]Preliminary research for this paper was presented at FTCS-20[23].

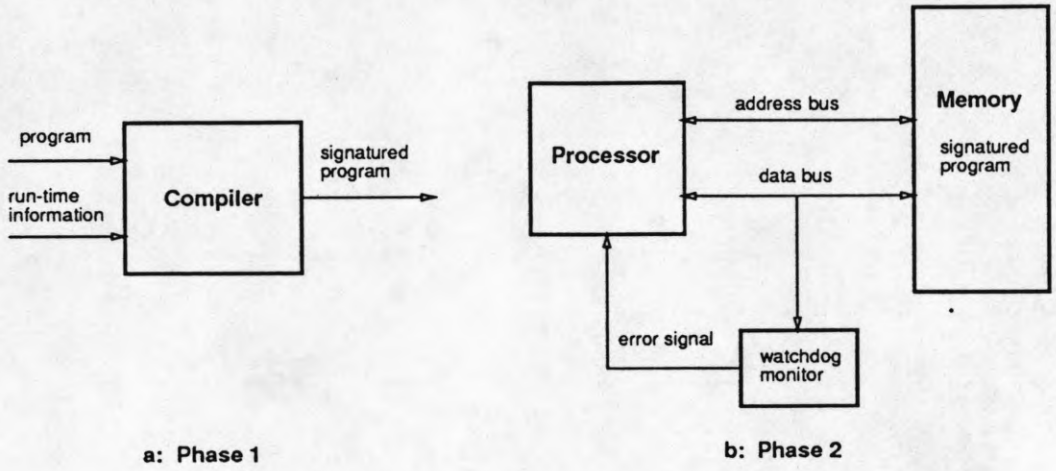


Figure 1: The phases of signature monitoring.

bound on the error detection latency.

To analyze the effectiveness of our compiler-assisted approach we compare the performance and memory overheads with the best hardware-assisted method, Wilken and Shen's Embedded Signature Monitoring [25]. In addition, we analyze the effect of bounding the error detection latency on the performance overhead, memory overhead, and error coverage.

2 Signature Monitoring

There are two phases to signature monitoring as shown in Figure 1. In the first phase, the compiler generates the signatures off-line and either embeds them into the original code [5, 10, 14, 17, 19, 20, 26] or provides the information directly to the watchdog [5, 15]. During the second phase, the watchdog monitor computes a run-time signature based on the instructions fetched by the target processor. At certain points the run-time signature is compared against the precomputed signature. Errors in the instructions or in their sequencing are detected if the signatures differ.

A program can be represented as a control flow graph. A typical control flow graph is presented

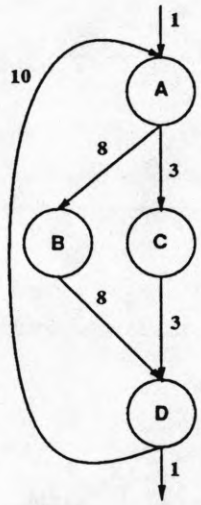


Figure 2: Weighted program control flow graph.

in Figure 2. A node represents a sequence of instructions with only one entry and one exit point. Arcs represent the flow of control as determined by branch statements. The weights on the arcs represent the execution frequency of that branch. For programs that are not self-modifying, the control flow graph is fixed and known at compile time. For compilers that can estimate the run-time behavior of the program, the weights are also known at compile time. This graph is used to generate signatures.

There are two types of signatures, reference and justifying. A reference signature is used to verify the control flow of a *program interval* which can consist of one or more nodes. Reference signatures are inserted either within the entry node or within the exit node of an interval. If it is inserted within the entry node of the interval, when the signature is fetched the watchdog performs a zero check on the run-time signature and resets the run-time signature to the new reference value. On the other hand, if it is inserted within the exit node of the interval, when the signature is fetched the watchdog verifies its run-time signature with the reference value and resets the run-time

signature to zero.

If an interval associated with a reference signature includes more than one node, the signature at either the branch or the merge point, for entry node and exit node insertion respectively, is inconsistent. Justifying signatures are used to make the signature consistent at these points. Justifying signatures can be inserted either within a node, *justifying node insertion*, or on an arc, *justifying arc insertion.*

2.1 Existing Approaches

Namjoo's Path Signature Analysis (PSA) is an example of node insertion [14]. In the original PSA, reference signatures are inserted at the beginning of each node. To reduce the memory and performance overhead, generalized PSA (Figure 3a) computes reference signatures for an interval or path set with a common start node. For each branch in the path set, the signatures will become inconsistent. Justifying signatures are added to make the signatures of all paths within a path-set consistent.

In more recent approaches, reference signatures are assigned to the exit or terminal nodes of paths. In such approaches, the signatures are inconsistent at the merge nodes. In the Signatured Instruction Stream (SIS) approach (Figure 3b) which uses Branch Address Hashing (BAH), reference signatures are placed before a merge on the sequential path [17, 18, 19]. Instead of using explicit justifying signatures, Shen and Schuette hash the branch address with the implicit signature value of the branch. If the run-time signature is incorrect then the rehashed branch address will be incorrect and the error will be detected unless the incorrect target is to another merge node. Although this scheme does not use justifying signatures, it is a predecessor of arc insertion because the implicit signature is only hashed along the taken arc of a branch.

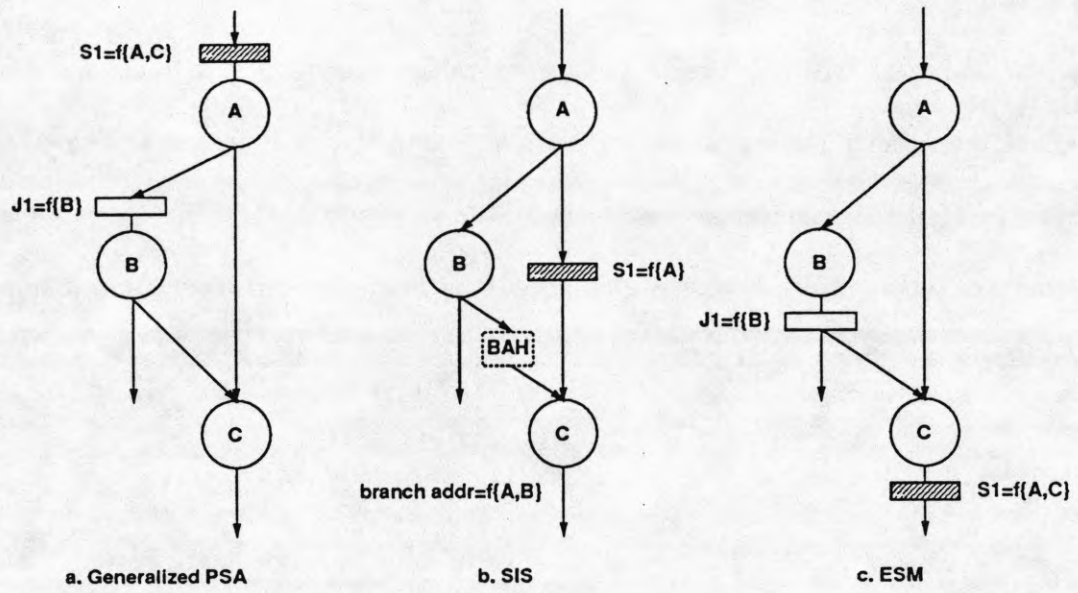


Figure 3: Existing signature monitoring schemes.

Embedded Signature Monitoring (ESM) is a hybrid node/arc insertion method (Figure 3c) [25, 26]. The compiler inserts justifying signatures within the node after a branch instruction. At run-time, hardware is used to determine whether or not the branch is taken. If it is then the justifying signature is included into the run-time signature. Otherwise it is discarded. Thus, the justifying signature is only included into the run-time signature along the taken arc of a branch.

In general, in arc insertion justifying signatures can be placed on any merge merge arc, not just the taken arc of a branch. Our signature model presented in Section 4.1.1 considers all of the cases for arc insertion.

2.1.1 Software Complexity

The implementation complexity includes both the hardware and software complexities. In this paper, the software complexity refers to the time required to compile a program. For a signature

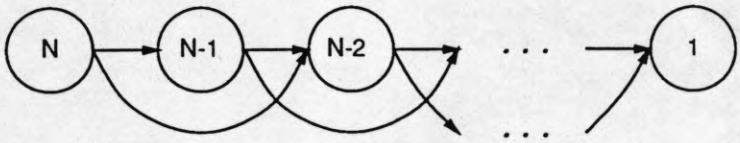


Figure 4: Directed acyclic graph with out-degree two.

monitoring approach to be practical the time to compile a program with signatures must be reasonable. The algorithm complexity of the signature insertion method reflects the additional time required to compile the program with signatures. In addition, any optimizing compiler techniques used specifically for signature insertion should also be included in the software complexity.

In Namjoo's PSA node insertion algorithm, all paths within a program interval are enumerated [14]. These paths are then resolved to determine the justifying signatures, their placement, and the reference signature of the interval. As shown in the following theorem, this algorithm has exponential complexity.

Theorem 1 *The maximum number of paths between two nodes in a directed acyclic graph with an out-degree of two is exponential in the number of nodes in the interval.*

Proof For the graph depicted in Figure 4 if node N is added to the graph with arcs to nodes *N-1* and *N-2* then the number of paths is $P(N) = P(N-1) + P(N-2)$. This is the Fibonnaci recurrence. The solution is

$$F_N = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^N - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^N. \quad \square$$

In Section 5 we present algorithms for arc insertion which have $O(N^2)$ complexity for a program graph with N nodes. In addition, we discuss the software complexity associated with the optimizing compiler techniques we use.

2.1.2 Hardware Complexity

To reduce the performance overhead due to inserting the signatures into the program code, previous methods have used hardware assists. Namjoo modified PSA by moving the signatures from the program code to the Cerebus-16 watchdog monitor environment [15]. Eifert and Shen extended SIS by removing the signatures from the program code and instead storing the program control flow graph and signature information in the monitor memory [5]. This method, Asynchronous Signature Instruction Stream (ASIS) can monitor multiple processors continuously. Both of these schemes eliminate the performance overhead but significantly increase the monitor complexity.

SIS and ESM use simple hardware assists to reduce the number of signatures fetched by the processor and thus reduce the performance overhead. SIS uses branch detection and address hashing hardware to combine the signature with the branch instruction. ESM uses hardware to determine whether or not the branch is taken or not.

3 Monitor Architecture Model

The watchdog monitor design should be simple and easy to integrate into the target system. It is especially important to keep the monitor design simple if the target processor has an on-chip instruction cache. Since the monitor must lie between the processor and memory, the monitor will have to be integrated into the chip design. To simplify the monitor and ease integration, we assume that the signature placement scheme does not require additional hardware support or place restrictions on the target architecture.

The two basic parts of the monitor are the interface and checking modules. The interface module is responsible for detecting instruction words and signatures and propagating the error

signal from the checking module to the target processor. The interface module is target processor dependent. Previous work has addressed the interface implementation issues for a variety of target architectures [9, 14, 16, 18, 20].

The checking module is application specific rather than processor specific. The signature encoding scheme is chosen based on the error coverage, error detection latency, and performance and memory overhead requirements of the application. The basic functions of the checking module are to generate the run-time signature, encorporate justifying signatures, compare against reference signatures, and propagate an error signal to the interface module if the run-time and reference signatures disagree.

Subroutine calls and interrupts require special handling. Previous methods use signature stacks to store the signature during a subroutine call or interrupt handling routine [4, 5, 18, 19]. On a subroutine return or return from interrupt, the signature is popped off the stack and checking of the interrupted routine continues. The signature stack significantly increases the monitor complexity because it requires a memory interface to handle stack overflows. Saxena and McCluskey propose a software approach for target processors that support coprocessors [16]. On an interrupt, the signature can be saved by generic processor save/restore routines. While this simplifies the monitor complexity, it will increase the performance and memory overheads. Wilken and Shen eliminate the signature stack by using a characteristic signature for each routine [26]. On a return from interrupt, this characteristic routine is used to justify the run-time signature. The disadvantage of this approach is that reference signatures cannot be inserted within the interrupt handling routines.

In our approach, we assume that there is a bound on the error detection latency. If the error is not detected within this bound, the error is assumed to be undetected. If a signature stack is used and an error occurs within a program interval before an interrupt, the error will not be detected

until after the interrupt handler has been executed. Such errors will likely exceed the bound on the error detection latency and are considered undetected. Therefore, signature stacks are not included in our model. To eliminate the need for a subroutine signature stack, we assume that reference signatures are placed before a subroutine call and at the end of a subroutine.

Interrupts, on the other hand, are asynchronous and therefore reference signatures cannot be placed before an interrupt. Instead, the signature checker is reset on an interrupt and checking begins on the interrupt handling routine. Reference signatures are inserted within the handling routine in order to satisfy the bound and at the end of the routine. On a return from interrupt, the signature checker is disabled until the next reference signature is fetched. After that normal checking resumes.

The elimination of signature stacks greatly simplifies the monitor hardware. In addition, for on-chip monitors the signatures do not need to be incorporated into the processor state. Therefore, it is possible to integrate the monitor without major modifications to the original processor design.

4 Signature Insertion Model

The signature insertion model indicates how justifying signatures and reference signatures should be inserted into the program code in order to guarantee that the program is properly encoded. Furthermore, the justifying signature insertion model is designed to minimize the performance overhead and the reference signature insertion model is designed to guarantee a specified bound on the error detection latency. The models have low software complexity and do not require special hardware support beyond the basic monitor.

4.1 Justifying Signature Insertion

In this section we present our arc insertion model and show how optimizing compiler techniques can be used to simplify the monitor and reduce the performance overhead.

In justifying arc insertion, the program interval is justified at the program merge nodes. At a merge node, the signature along each incoming arc is different. Only one signature can be used to define the signature at the merge node. Justifying signatures are used to transform the remaining incoming signatures to this unique signature. There is only one constraint to placing the signatures on the program arcs.

Constraint 1: For a merge node with i incoming arcs, justifying signatures must be placed on $i - 1$ arcs.

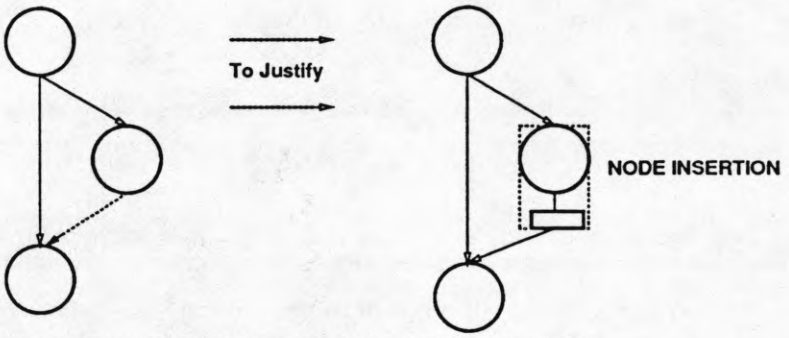
The arcs with justifying signatures are *justifying arcs* and the remaining arc is the *unique arc.*

4.1.1 Arc Insertion Model

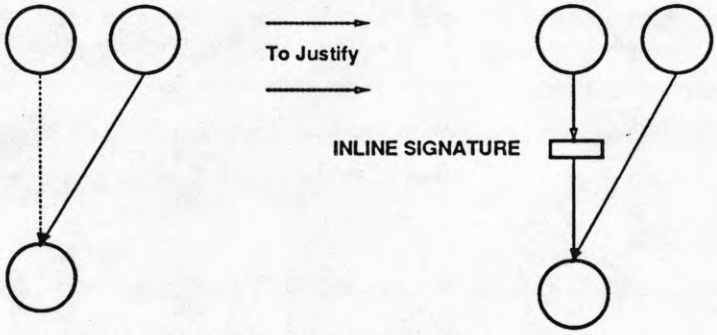
There are three types of justifying arcs, which are drawn as dashed lines in the control flow graphs of the three cases in Figure 5.

In the first case, the justifying arc represents an unconditional branch. Since it is an unconditional branch, the signature can be placed directly in the node without affecting any other program path. The signature can either be placed before or after the branch instruction. If the target architecture always fetches the instruction following a branch, it can be placed after the branch. Otherwise, it must be placed before the branch.

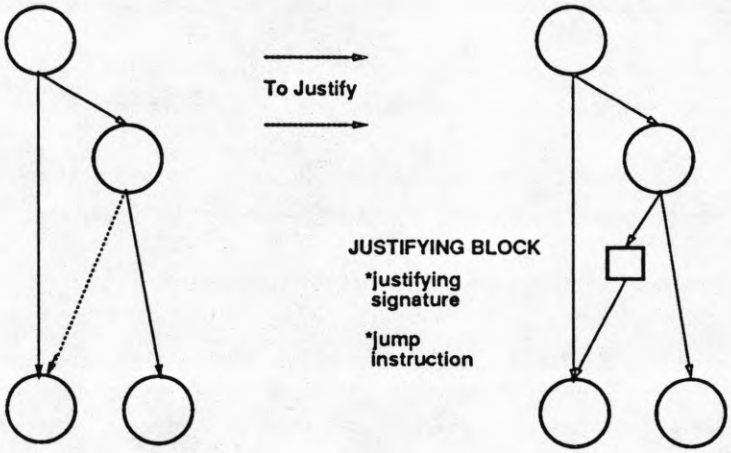
In the second case, the justifying arc is on the sequential path. The sequential path can either be the not taken path of a conditional branch or after a non-branching node. Either way, the last



CASE 1: JUSTIFYING ARC FROM UNCONDITIONAL BRANCH



CASE 2: JUSTIFYING ARC ON SEQUENTIAL PATH



CASE 3: JUSTIFYING ARC IS THE TAKEN PATH OF CONDITIONAL BRANCH

Figure 5: Justifying arc insertion.

instruction in the source node and the first instruction in the destination node of the justifying arc are in sequential memory locations. The justifying signature is placed between these two instructions.

In the third case, the justifying arc is on the taken path of a conditional branch. In this case the source and destination nodes of the justifying arc are not in sequential memory locations. Therefore, to place the justifying signature on the arc, a *justifying block* is inserted between the source and destination nodes. The justifying block consists of a signature instruction and a jump instruction. The destination of the branch instruction in the source node is modified to jump to the justifying block, and the justifying block jumps to the original destination node.

4.1.2 Justifying Signature Generation

For arc insertion, signature generation depends on the following property.

Property 1: There is a path along unique arcs between the start and terminal nodes of a program interval.

Based on this property, all of the signatures of the unique arcs in a program interval can be determined by a breadth first search. After all the unique arcs are labeled with their signatures, the justifying signatures can be generated as shown in Figure 6. The justifying signature J1 is a function of the unique signature S_i, the unique signature of its source node S_j, and the signature of node A.

4.1.3 Optimizing Compiler Techniques

In an optimizing compiler, the architectural features of the target processor are known so that the compiler can order the instructions such that they fully utilize the target processor while not

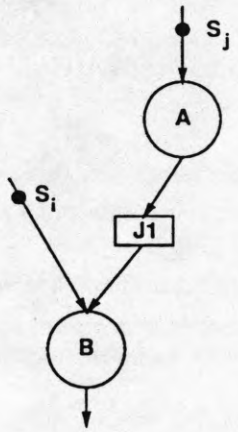


Figure 6: Signature generation for arc insertion.

violating the execution order. In a similar fashion, the target processor features can be used to ensure that signatures are placed properly. That is, only signatures that are supposed to be included into the run-time signature are fetched by the target processor. In particular, the branch handling scheme must be accounted for. For example, recall that the MC68000 always fetches the instruction following the branch and discards it if the branch is taken. Therefore, signatures can always be placed after an unconditional branch without incurring any performance penalty. On the other hand, signatures cannot be inserted directly after the branch on the sequential arc. Otherwise, if the branch is taken then the signature will be incorrectly included into the run-time signature. A detailed performance and memory cost analysis for a variety of branch handling mechanisms is provided in Section 6.1.1.

Another optimizing compiler technique is to use run-time information to improve the processor performance. For instance, run-time information can be used to place instructions to improve sequential locality. Run-time information can also be used to place signatures to reduce the performance overhead. The minimum number of justifying signatures required to encode a program

interval with one reference signature and n conditional branches is n [26]. Arc insertion places the minimum number of signatures into the program code. Our goal is to use run-time information to minimize the number of signatures *fetched* and thus minimize the performance degradation.

In arc insertion, any merge arc can be selected as the unique arc. Run-time information can be used to guide this selection. By measuring the run-time behavior of the program, the node execution and branch frequencies can be predicted. Based on this prediction, the cost of inserting a signature on each merge arc can be determined. The cost, *arc_cost*, in terms of number of instruction words fetched, is:

$$arc\_cost = arc\_frequency * node\_weight * just\_words.$$

For example, if the signature is placed on the taken path of a conditional branch, *arc_frequency* is the probability that the branch is taken, *node_weight* is the number of times the branch is executed, and *just_words* is the number of instructions words required for a justifying block. The *just_words* also reflects cost of the special architectural features of the target processor.

The following theorem proves that using *arc_cost* to select the unique arc minimizes the performance overhead for justifying arc insertion.

Theorem 2 *If the unique arc of each merge node corresponds to the incoming arc with the highest arc_cost, the number of instruction words fetched to justify the program is minimized.*

Proof Since justifying signatures are placed on the arcs, the signature assignments for each merge node do not depend on the assignments at other merge nodes. Therefore, the total number of justifying signatures fetched is the sum of the justifying signatures fetched at each merge node. For a single merge node, if the unique arc has the highest *arc_cost* of all the incoming arcs, the number of instruction words fetched to justify that node is a minimum. Since a

sum of minimums is a minimum sum, the number of instruction words to justify the entire program is minimized. □

This theorem proves that using run-time information will minimize the performance overhead for justifying arc insertion. In the experiment section (Section 6) we empirically prove that optimized arc insertion (i.e., using run-time information) minimizes the overhead due to justifying signatures.

4.2 Reference Signature Insertion

The separation of reference signatures defines the checking interval l_{max}. For bit errors, the average detection latency is $l_{max}/2$ and the maximum detection latency is l_{max} [26]. For single sequence errors, the average detection latency is l_{max} and the maximum detection latency is $2l_{max}$. Let B be the bound of the error detection latency for all bit errors and single sequence errors. Reference signatures must be placed such that l is at most $B/2$.

4.2.1 Reference Insertion Model

The reference signature insertion model is shown in Figure 7. A reference signature is required at each program exit point in order to correctly check the program (case 1). Recall that a signature stack will violate the bound on the error detection latency. To eliminate the need for a signature stack for subroutine calls, reference signatures are placed before the call and at the end of the routine (cases 2 and 3). A reference signature is placed at the end of an inner loop, case 4, in order to guarantee that loops of length less than l_{max} do not violate the bound on the detection latency. Furthermore, this breaks cycles in the program graph which simplifies the reference placement algorithm presented in the next section. Finally, signatures are placed such that no two are farther apart than l_{max} (case 5).

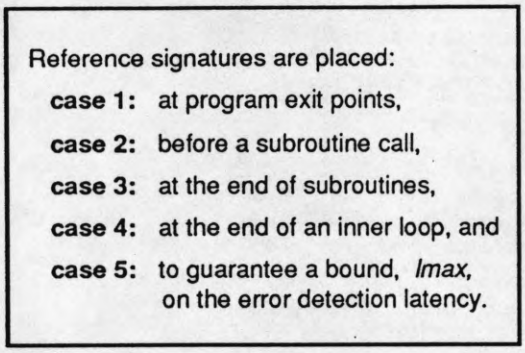


Figure 7: Reference signature insertion model.

5 Signature Insertion Algorithms

In this section, the algorithms for placing and generating both justifying and reference signatures are presented. A discussion of the algorithm complexities and overhead associated with collecting run-time information is provided at the end of the section.

5.1 Justifying Signature Placement Algorithm

The algorithm for justifying signature placement[4] is shown in Figure 8. The algorithm implements the justifying arc insertion model and generates a partial terminal node set T. This set corresponds to the first four cases of the reference signature model, namely, a program or subroutine exit node, an inner-loop exit node, or the node before a subroutine call. The program control flow graph, G, is the input to the algorithm. First, the terminal nodes are determined. Then, for each merge node, if all incoming arcs are from terminal nodes, none of the signatures need to be justified. Otherwise, a unique arc is selected. The unique arc can be specifically selected (e.g., using run-time information)

[4] For the algorithms in this section, it is assumed that the compiler converts all switch statements into the equivalent if-else construct, and program placement information is available at compile time to determine the taken path of a conditional branch.

```
/* place_justifying_signatures
   input: G = program control flow graph
   output: program graph with justifying signatures and
           partial terminal node set T                    */

place_justifying_signatures(G)
{
  for each node n in G
    if n is a terminal node
      add n to the terminal node set T
      place a reference signature at the end of n
  for each merge node m in G
    if all incoming arcs to m are from terminal nodes
      mark all arcs as unique
    else
      select a unique arc
    for each non-unique merge arc x
      if x from an unconditional branch
        place a justifying signature before the branch instruction
      else if x between two sequential nodes s1 and s2
        create a justifying signature after the last instruction of node s1
      else                    /* x is the taken arc of a conditional branch */
        create a justifying block and place between the conditional
          branch node and the target node
        correct the target labels
}
```

Figure 8: Justifying signature placement algorithm.

or it can be selected at random. Note that for an unconditional branch, the signature can be placed after the branch for target architectures that always fetch the signature following a branch. The MC68000 is an example of such an architecture [3].

5.2 Reference Signature Placement Algorithm

The algorithm for reference signature placement is shown in Figure 9 and its functions are shown in Figure 10. The algorithm places reference signatures so that the maximum distance between any two reference signatures is less than *lmax*. The program control flow graph, G, is effectively an acyclic graph since the terminal nodes break cycles. S and T represent the start node and terminal node set.

The algorithm is a greedy algorithm. Starting from the start node and each terminal node, it

traverses the paths of all successors calculating the maximum path length (step 1). The traversal along each path stops at a terminal node. When the successor of a node makes the path length greater than *lmax*, the current node is marked as a terminal node. The successors of the new terminal nodes are also traversed. The algorithm stops when all arcs have been visited. The reference signatures are then placed at the end of each terminal node (step 2).

During the traversal, when paths merge they are combined in *add_queue* into one path with the path length set to the maximum path length. In addition, the number of duplicates of the end node, *dups*, is incremented. A merge node is only removed from the queue in *remove_queue* when all incoming paths have been traversed (i.e., *p.dups* is equal to the number of predecessors of *p.end_node*).

5.3 Signature Generation Algorithm

The signature generation algorithm is presented in Figure 11. Unique arcs have been identified by the justifying signature placement algorithm. The unique intermediate signatures are marked using a breadth first search. Once all the unique arcs have been marked with their intermediate signatures, the reference signatures are known and the justifying signatures can be calculated as shown in Figure 6 in Section 4.1.2.

5.3.1 Complexity Analysis

For a program graph of N nodes, the complexity of the justifying signature placement algorithm is $O(N^2)$. To generate the terminal nodes, loop analysis must be performed. The complexity of the loop generation algorithm is $O(N^2)$[1]. Once loop analysis has been performed, N nodes are considered to identify and mark the terminal nodes. To mark the unique arcs, at most $2N - 2$

```
/* place_reference_signatures
     inputs: G = program_graph, S = start node,
             T = partial terminal node set,
             lmax = 1/2 error detection latency bound
    outputs: program graph with reference signatures
             placed no further apart than lmax and
             the complete terminal node set T          */

place_reference_signatures(G, S, T, lmax)
{
  p = generate_path(S)                                  /* step 1 */
  add_queue(ref_queue, p)
  for each terminal node t in T
    for each successor s of t
      p = generate_path(s)
      add_queue(ref_queue, p)
  while ref_queue not empty
    p = remove_queue(ref_queue)
    if p.length + max(|successors of p.end_node|) > lmax
      mark p.end_node as a terminal node and add to T
    for each successor s of p.end_node
      if p.end_node is a terminal node
        new_p = generate_path(s)
      else
        new_p = update_path(s,p)
      add_queue(ref_queue, new_p)
    destroy p
  for each node in G
    if a terminal node                                  /* step 2 */
      place a reference signature at the end of the node
}
```

Figure 9: Reference signature placement algorithm.

```
/* generate_path
     input:  n = program graph node
     output: path p which has path length
             equal to length of n,  n,
             number of duplicates of n
             initialized to 1                */

generate_path(n)
{
  create p
  p.length = |n|
  p.end_node = n
  p.dups = 1
  return p
}

/* update_path
     inputs:  n = program node,  p = current path
     output:  a new path, new_p, which has path
              length set to length of p + length of n,
              n, the number of duplicates of n
              initialized to 1                    */

update_path(n)
{
  create new_p
  new_p.length = p.length + |n|
  new_p.end_node = n
  new_p.dups = 1
  return new_p
}

/* add_queue
     inputs:  queue = list of paths,  p = path to add
     output:  queue with either a new path p or an
              updated path e that has the same end
              node as p.  the updated path e has length
              set to the maximum length of p and e and
              the number of duplicates of the end node
              of e is incremented                 */

add_queue(queue, p)
{
  for each element e in queue
    if e.end_node = p.end_node
      e.length = max(e.length, p.length)
      e.dups = e.dups + 1
    else
      add p to end of queue
}

/* remove_queue
     input:  queue = list of paths
     output:  path p whose end node has had
              all its incoming arcs visited    */

remove_queue(queue)
{
  p = first element of queue
  while p.dups != number of predecessors of p
        add p to end of queue
        p = first element of queue
  return p
}
```

Figure 10: Functions of the reference signature placement algorithm.

```
/* signature_generation
   inputs: G = program graph, S = start node,
           T = terminal node set
   output: program graph with signatures
           generated                          */

signature_generation(G,S,T)
{
for each node n in {S,T}
    for each successor s of n
        if s is an unmarked unique arc
          mark the intermediate signature on the
           unique arc
          push s on unique_stack
    while unique_stack not empty
          pop n off unique_stack
          if n is not a terminal node
            for each successor s of n
                if s is an unmarked unique arc
                  mark the intermediate signature
                   on the unique arc
                  push s on unique_stack
            else
              calculate the reference signature of n
for each merge node in G
    calculate the justifying signature of the non-unique
     incoming arcs
}
```

Figure 11: Signature generation algorithm.

merge arcs are considered for a graph with N nodes and an out degree of two.

The complexity of the reference signature placement algorithm is also $O(N^2)$. In step 1, *add_queue* is called once for each arc and *remove_queue* is called once for every node other than the initial start and terminal nodes. Both *add_queue* and *remove_queue* linearly search the queue and thus have $O(N)$ complexity. In step 2, each node is evaluated once. Therefore, the reference algorithm has $O((2N-2)*N) + O(N^2) + O(N) = O(N^2)$ complexity.

In the signature generation algorithm, the intermediate signature of each arc is marked once. Therefore, it has $O(N)$ complexity. The complexity of all the algorithms combined is $O(N^2)$. Compared to the exponential complexity of justifying node insertion, $O(N^2)$ complexity makes justifying arc insertion a desirable approach.

If run-time information is used to select the unique arcs, the performance overhead due to

justifying signatures can be minimized. If a profiler is used to collect the run-time information, the program is run for a variety of inputs while the execution frequencies are calculated. Therefore, the time to compile increases. However, for production code, this one-time cost may be worth the improved performance.

6 Experimental Results

In this section, we present the results of experiments performed to study the performance of compiler-assisted arc insertion and hardware-assisted node insertion and to analyze the impact of bounding the error detection latency.

To perform the experiments, we added profiling and signature placement to the GNU C compiler. Programs were compiled with probes inserted at each node. At run-time these probes were used to collect the branch and node execution frequencies. These frequencies, combined with the architecture specifications, were used to guide signature placement. Thus, the complete process for inserting signatures is to compile the program with probes, profile the program on a large set of sample inputs, and re-compile the program to place signatures.

The experiments were performed using the benchmark set shown in Table 1. The ten benchmarks[5] are a combination of Unix, CAD, and text processing programs. The largest benchmark is more than an order of magnitude larger than benchmarks of previous studies [17, 19]. The sizes of the input sets used in profiling are also given in Table 1. The average node or basic block size of each benchmark is given for the MC68000.

[5]These benchmarks are control intensive. Results for numerical applications will be better.

Benchmark	Description	Size (bytes)	Number of inputs
cmp	file comparison	2406	16
compress	compress/expand files	14410	20
diff	file comparison	32314	19
eqn	format equations	55175	20
grep	search file for expression	4630	20
mpla	tile based PLA generator	24104	19
tar	create tape archives	22612	14
tbl	format tables	65117	21
wc	line/word/char count	1686	20
yacc	parsing program generator	48444	10

Table 1: Benchmark characteristics.

6.1 Performance of Arc Insertion

In this section we compare the performance and memory overheads of a compiler-assisted arc insertion and a hardware-assisted node insertion scheme for a variety of branch handling methods. Justifying Arc Insertion (JAI) is our arc insertion which uses the algorithm in Section 5.1. In Optimal JAI, the signatures are placed using run-time information. In Random JAI, each unique signature is randomly selected.

Wilken and Shen's Embedded Signature Monitoring (ESM) scheme is a hybrid node-arc insertion method. It has the performance and memory overheads of node insertion but the software complexity of arc insertion. Signatures are placed within a node after a branch instruction. At run-time, hardware is used to determine if the branch is taken. If so, the signature is included into the run-time signature; otherwise, it is discarded. Therefore, signatures are generated to justify the arcs.

For these experiments, there was no tight bound on the error detection latency (in the next section, the effects of bounding the error detection latency are presented). Reference signatures were placed at the program and subroutine exit nodes, before subroutine calls, and at the inner-loop

exit nodes. In order to make the schemes comparable, these signatures were also inserted for ESM, which originally only inserts reference signatures at the program exit nodes.

When a signature is placed after a branch, some or all of the performance overhead may be hidden by the branch handling behavior of the target architecture. We ran our experiments for three branch handling schemes: prefetch, delayed branching, and Branch Target Buffer (BTB). In the prefetch scheme, the instruction following the branch is always fetched. If the branch is taken, the instruction is discarded. The MC68000 uses this branch handling method [3]. For delayed branching, we assume that the delay slot can be filled 70% of the time for a conditional branch and 100% of the time for an unconditional branch [13]. In the BTB scheme, the expected target for the branch is fetched from the buffer. We assume that if the target is wrong, the correct target is determined within one instruction cycle [8].

6.1.1 Cost Analysis

The performance and memory cost of inserting a signature depends on the insertion scheme and the target processor architecture. In ESM, a hardware monitor is used to determine whether the branch is taken or not. This hardware depends on the branch handling hardware of the target processor. In JAI, the hardware monitor is kept independent of the processor architecture and implementation by using this information at compile-time to place the signatures. To guarantee correct checking, the signatures must be placed such that the monitor does not see any incorrectly fetched signatures. In some cases, to ensure independence from the basic system architecture, NOP instructions are added which increase the performance and memory overhead costs.

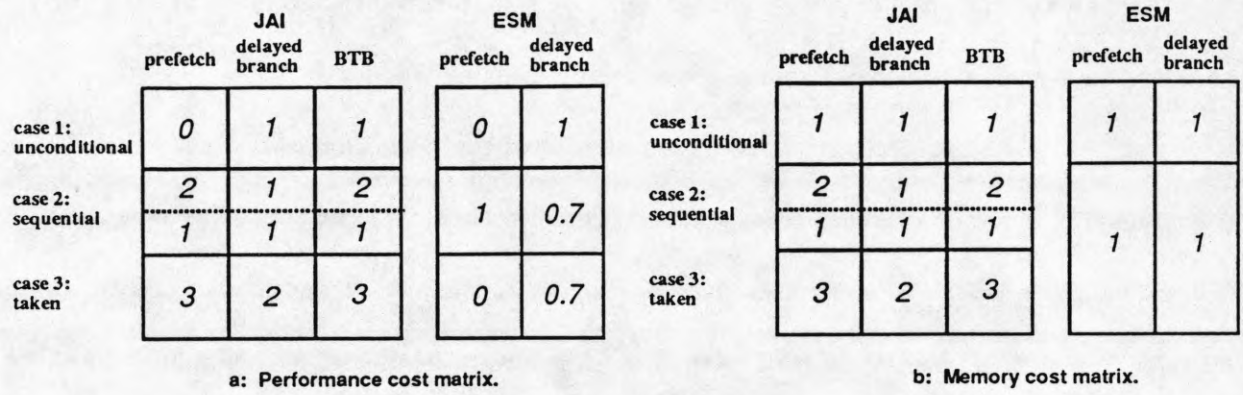


Figure 12: Performance and memory cost matrices.

The performance and memory overhead costs are presented in Figure 12 [6] [7]. The cost depends on the signature insertion scheme and the branch handling method. The three cases correspond to the three cases in the arc insertion model in Figure 5 in Section 4.1.1. ESM was not designed to work with a BTB and thus the cost for this combination is not presented [8]. For all of the cases we assume that the justifying signature instruction requires one instruction word. The number of instruction words required to implement the justifying block is discussed in case 3.

Performance cost. The performance cost matrix in Figure 12a indicates the number of instruction words fetched per justifying signature. Each case is described in detail below.

case 1 - unconditional branch: For an unconditional branch, both schemes place the signature in the node. The signature is placed after the branch for the prefetch scheme and delayed branching. Since the instruction after an unconditional branch is always discarded in the prefetch scheme, the cost for both JAI and ESM is zero. In delayed branching, the delay

[6] The matrices reflect the cost when the corresponding arc types are traversed. Optimal JAI traverses fewer arcs than ESM and thus has a lower overall cost.

[7] The cost for JAI depend on the location of the monitor. The costs presented are conservative. If the monitor is placed after the instruction register then the cost will be lower.

[8] The ESM hardware monitor could be modified to handle a BTB based target processor.

slot can always be filled for an unconditional branch and thus the cost is one. For the BTB scheme, the signature is placed before the branch and thus the cost is one.

case 2 - sequential path: For JAI, if the sequential path corresponds to the not taken path of a conditional branch (top number for case 2 in Figure 12a), the signature cannot be placed directly after the branch. For the prefetch and BTB schemes, to guarantee that the signature is not included when the branch is taken, a NOP instruction is inserted between the branch and the signature and thus the cost is two. This is a cost paid to insure that the monitor is independent of the basic system architecture.[9] For delayed branching, the delay slot is filled from before the branch for a conditional branch. Therefore, the signature is placed after the delay slot and the cost is one. If the sequential path does not correspond to the not taken path of a conditional branch, the cost is one for all of the branch handling methods.

In ESM, the signature is always fetched and discarded by the monitor if a conditional branch is not taken. For the prefetch scheme, the instruction following the branch is always executed if the branch is not taken. Therefore, the cost is one. In delayed branching, the delay slot can be filled 70% of the time and thus the cost is 0.7.

case 3 - taken path: For JAI, justifying blocks are placed on the taken arc of the conditional branch. For prefetch and delayed branching, the signature is placed after the jump instruction in the justifying block. For the BTB method, the signature is placed before the justifying block jump instruction. To prevent the signature from being included into the run-time signature when the target of the conditional branch in the BTB is incorrect, a NOP instruction is inserted before the signature in the justifying block. Again, this cost is the result of

[9] It will be shown in Figure 14 that this cost is not incurred in practice.

insuring system architecture independence. The cost is simply the number of instruction words required for the justifying block. We assume that the justifying block size is three instruction words for the prefetch and BTB schemes, and two instruction words for delayed branching.[10]

For ESM, the signature is placed directly after the branch. For the prefetch scheme, the instruction is squashed if the branch is taken and thus the cost is zero. For delayed branching, the delay slot can be filled 70% of the time and thus the cost is 0.7.

Memory cost. The memory cost matrix in Figure 12b indicates the number of instruction words inserted into the program code. Note that cases 2 and 3 for ESM actually stem from one signature being placed after a conditional branch. Therefore, the memory cost of the cases combined is one.

6.1.2 Performance Overhead

In this section we present the relative performance overhead results for Optimal JAI, Random JAI, and ESM for the three branch handling methods. We also present the performance overhead of the three insertion schemes for the MC68000 target processor.

Figure 13 shows how each insertion scheme performs relative to Optimal JAI for each branch handling method. As can be seen, Optimal JAI has the minimum performance overhead for all of the branch handling schemes. However, for the prefetch branch method, Random JAI performs almost as well as Optimal JAI. Since the cost of an unconditional branch is zero for the prefetch scheme, it appears that Random JAI places most of its signatures on the unconditional path. The

[10]The prefetch scheme estimate is based on the MC68000 which needs two instruction words for a jump instruction. For the others we assume one word per instruction.

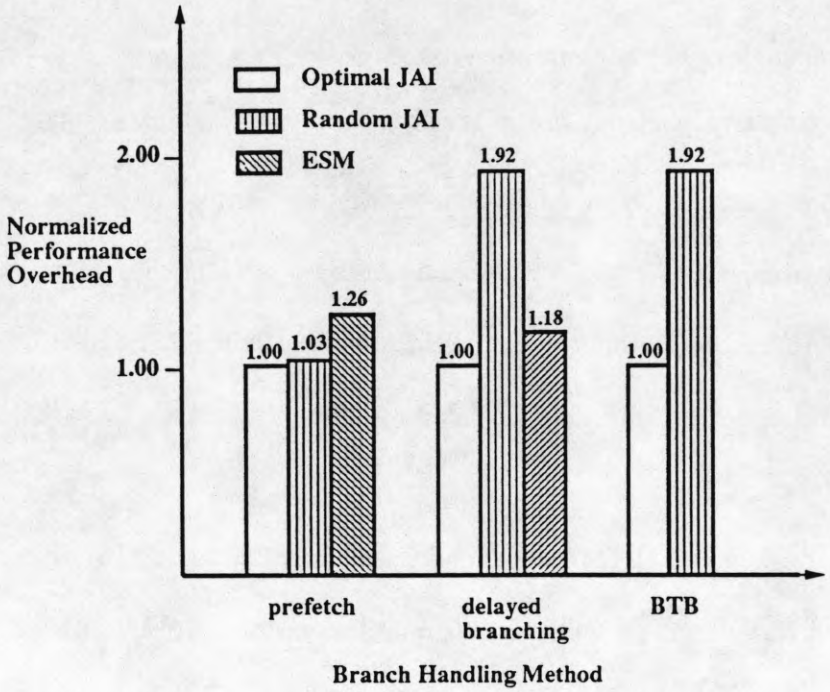


Figure 13: Normalized performance overhead.

graph in Figure 14b showing the distribution of the performance overhead for Random JAI confirms this conclusion. Note that for the delayed branching and BTB methods, Optimal JAI adjusts for the cost of an unconditional branch whereas the signature placement in Random JAI does not change. Therefore, for these two branch handling methods, the performance overhead for Random JAI is almost double the performance overhead of Optimal JAI.

For all the schemes and branch handling methods, the number of reference signatures inserted is the same. Therefore, the relative percentage of performance overhead due to reference signatures (Figure 14) indicates the overall performance of the schemes for a given branch handling method. That is, the higher the percentage due to reference signatures, the better the scheme. For all of the signature insertion schemes, the percentage due to reference signatures shows that a processor with

prefetch branch handling will have the lowest performance overhead and processors with delayed branching will perform slightly better than processors with BTBs.

The distribution of performance overhead for ESM in Figure 14c shows the disadvantage of node insertion. In node insertion, for a conditional branch, signatures are fetched along both the taken and not taken (sequential) paths. For ESM, the signatures fetched along the sequential path account for 18.7% of the performance overhead for prefetch and 12.3% for delayed branching. These signatures are discarded in ESM but still incur a performance penalty. In arc insertion, these signatures are not fetched at all.

The performance overhead for the MC68000 in Table 2 shows that adding signature monitoring to an MC68000 based target system will only degrade the performance by approximately 4%[11]. This includes the overhead due to reference signatures placed before a call, at the subroutine and program exit nodes, and at inner-loop exit nodes. If the overhead due to these reference signatures is removed so that the program is only checked at the exit nodes, the performance overhead is reduced to approximately 0.1%. In this case, the error detection latency is the entire program execution time.

6.1.3 Memory Overhead

Figure 15 shows the normalized memory overhead for all of the branch handling methods. The same number of signatures were added for all insertion schemes. The difference in the memory overhead is due to the addition of justifying blocks. Since ESM does not use justifying blocks it has the lowest memory overhead. Instead, it uses additional hardware. Therefore, there is a tradeoff between the memory and hardware overheads. The fact that the memory overhead for Random

[11] The arithmetic mean is used to summarize the benchmarks [6].

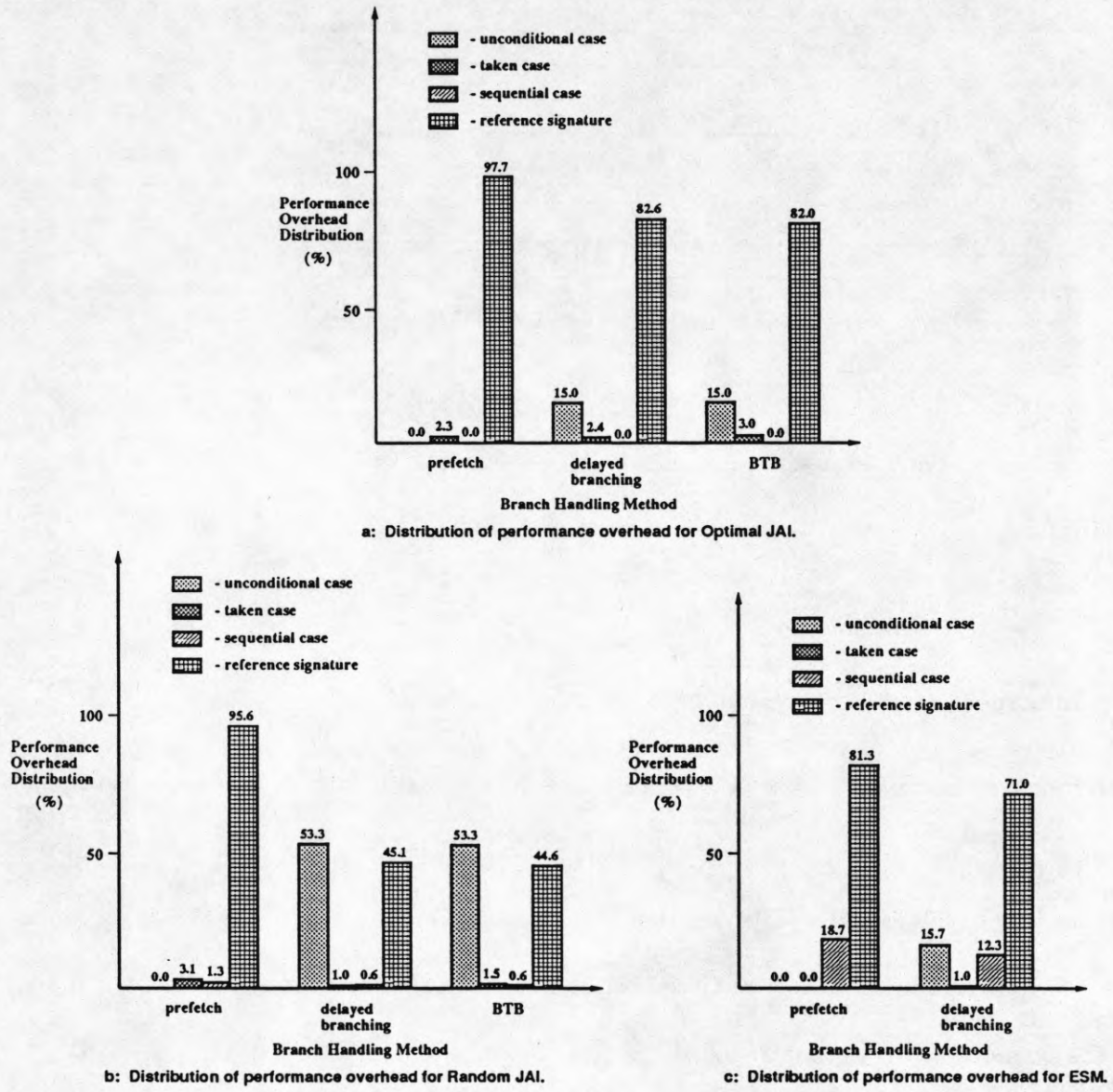


Figure 14: Performance overhead distributions.

Benchmark	Optimal JAI	Random JAI	ESM
cmp	1.80	1.80	1.80
compress	1.54	1.54	1.75
diff	3.05	3.08	3.72
eqn	3.81	4.02	6.08
grep	4.79	5.14	8.13
mpla	2.26	2.32	2.41
tar	6.70	6.70	7.20
tbl	6.44	6.61	7.23
wc	3.51	3.51	5.42
yacc	4.71	4.87	4.81
mean	3.86	3.96	4.86
std. dev.	1.80	1.85	2.36

Table 2: Percentage of performance overhead for the MC68000.

JAI is less than for Optimal JAI shows that there is also a tradeoff between the performance and memory overheads. The memory overhead for the MC68000 is shown in Table 3. On average there is approximately 11% memory overhead associated with adding JAI to a MC68000 based target system.

6.2 Bounding the Error Detection Latency

In this section we analyze the effect on the performance and memory overheads of varying the bound on the error detection latency. We also discuss the impact of reference signature placement on the error coverage. For this analysis, justifying signatures were optimally placed using the algorithm in Section 5.1. Reference signatures were placed using the greedy algorithm presented in section 5.2, where the bound $lmax$ is the maximum distance between two reference signatures. The target processor in the experiments was the MC68000.

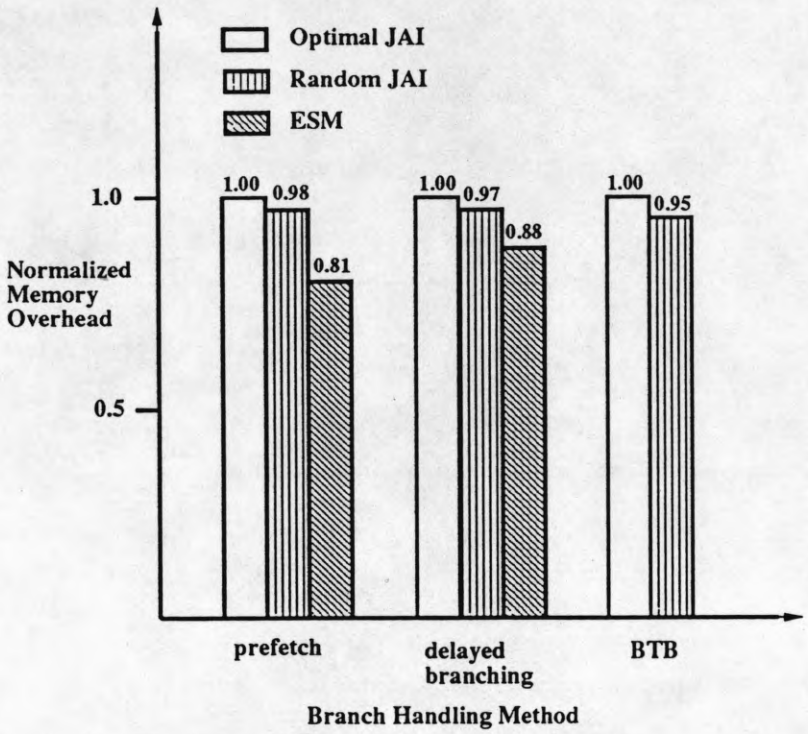


Figure 15: Normalized memory overhead.

Benchmark	Optimal JAI	Random JAI	ESM
cmp	13.25	13.25	9.52
compress	8.20	8.20	6.46
diff	10.56	10.53	8.60
eqn	8.65	8.72	7.46
grep	16.25	15.82	13.48
mpla	6.76	6.76	5.98
tar	9.57	9.64	8.52
tbl	12.34	12.11	9.90
wc	13.77	12.58	10.18
yacc	9.49	9.47	8.24
mean	10.88	10.71	8.83
std. dev.	2.94	2.72	2.14

Table 3: Percentage of memory overhead for the MC68000.

6.2.1 Performance Overhead

To study the effect of the error detection latency on the performance overhead, $p_{overhead}$, signatures were placed for 19 values of l_{max} [10, 20, ..., 100, 200, ..., 1000]. For each level of l_{max} there were 10 overhead observations (one for each benchmark). Assuming a normal distribution at each level of l_{max}, a non-linear regression analysis on the experimental observation yields the following statistical relationship:

$$p_{overhead} = 14.998e^{-0.049l_{max}} + 4.017.$$

The regression curve for performance overhead is shown in Figure 16. A plot of the residuals shows that the actual data points are evenly distributed around the predicted function and thus the fit is reasonable. For low values of l_{max} there is a significant change in the overhead for small changes in l_{max} until l_{max} is approximately 80 instructions. The worst case corresponds to placing signatures at each basic block. For a basic block length of 5 instructions[12] the worst case mean performance is approximately 15.76%. The asymptote of this relation, 4.017%, is the performance overhead due to justifying signatures and reference signatures placed at the subroutine exit nodes, inner-loop exit nodes, and before a subroutine call. For all of the benchmarks this asymptote is reached by $l_{max} = 300$ instructions.

The 95% confidence intervals for the expected value and for an individual prediction are also shown in Figure 16. From the individual prediction confidence interval, we can conclude that 95% of new programs will have a performance overhead between approximately 1% and 8% for an l_{max} of 100. Furthermore, the expected value confidence interval indicates that for $l_{max} = 100$, 95% of the time the mean value after including a new program will remain approximately between 3.5%

[12]The average basic block size for the benchmark set is 5.53 instructions.

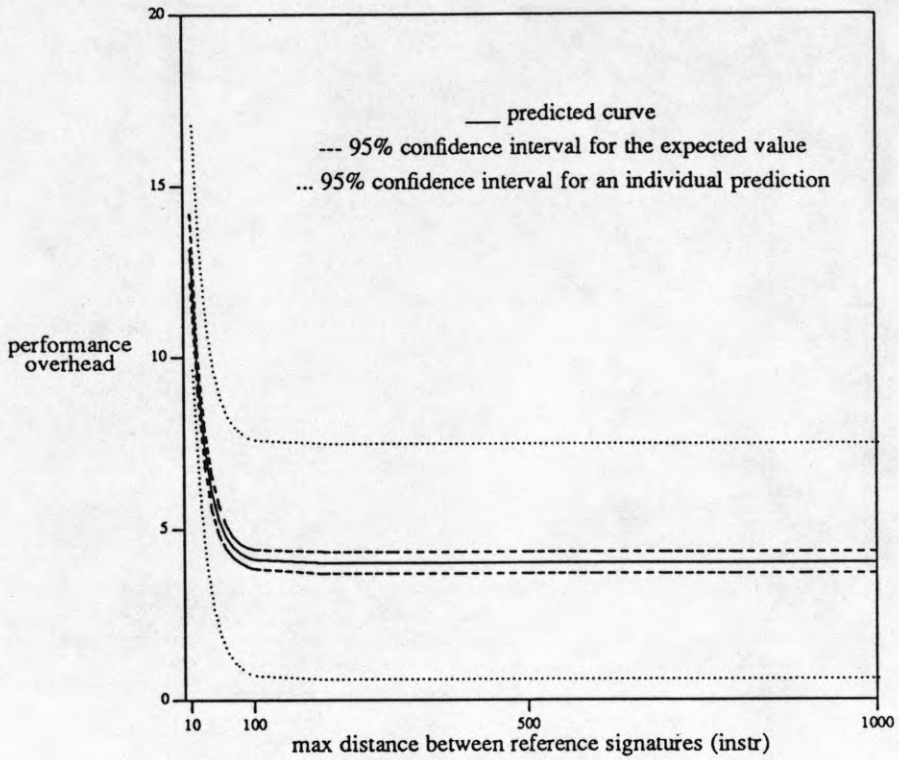


Figure 16: Predicted performance overhead with 95% confidence intervals.

and 4.5%.

6.2.2 Memory Overhead

The same experiments were performed to study the statistical relationship between the memory overhead, $m_{overhead}$, and l_{max}. Assuming a normal distribution at each level of l_{max}, a non-linear regression analysis on the experimental observations yields the following statistical relationship:

$$m_{overhead} = 7.848e^{-0.040l_{max}} + 10.927.$$

The regression curve with the 95% confidence intervals is shown in Figure 17. Again, the residual plot shows that the predicted curve is a good fit.

The worst case mean memory overhead ($l_{max} = 5$) is approximately 17.35%. The asymptote overhead is 10.927%. Note that the maximum difference between the performance overhead across the l_{max} range is approximately 12%. For the memory overhead, the maximum difference is approx-

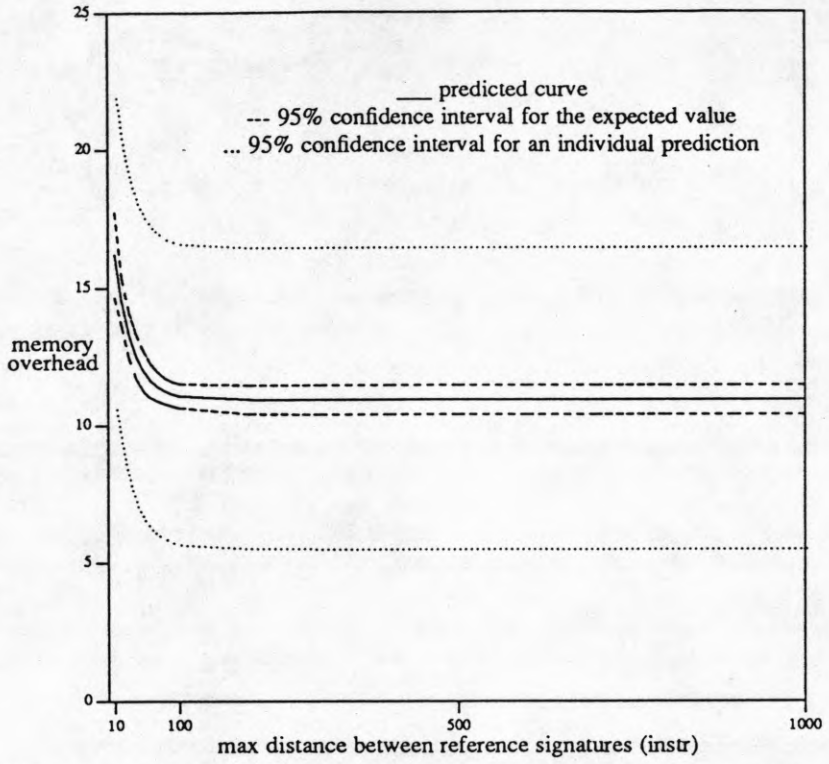


Figure 17: Predicted memory overhead with 95% confidence intervals.

imately 7%. Therefore, varying the error detection latency has greater impact on the performance overhead than on the memory overhead. On the other hand, the memory overhead, even for the worst case, is worse than the performance overhead. This implies that the longer basic blocks get executed more frequently.

From the confidence intervals of Figure 17, for an l_{max} of 100, 95% of the time a new program will have a memory overhead between approximately 6% and 17%, and the overall mean will remain between approximately 10.5% and 11.5%.

6.2.3 Error Detection Latency

For a bound l_{max}, the upper bound on the detection latency for double bit errors is l_{max}. If errors are evenly distributed, the average detection latency for bit errors is $l_{max}/2$. For single sequence errors, the maximum detection latency is $2l_{max}$. Therefore, for all single errors, the maximum

detection latency is $2l_{max}$. Near optimal performance can be achieved for $l_{max} = 100$ instructions or a maximum detection latency of 200 instructions.

6.2.4 Error Coverage

Consider a program interval of l_{max} w-bit instruction with a w-bit signature. Using Carter's MISER, all double bit errors are detected if:

$$l_{max} < \lfloor \frac{2^{\frac{w}{2}} - 1}{w} \rfloor \left(2^{\frac{w}{2}} + 1\right),$$

where w is the signature width [2]. For the MC68000, $w = 16$ and thus l_{max} must be less than 4112 to detect all double bit errors. Therefore, the bit error coverage will not be affected by varying l_{max} since there is no point in increasing l_{max} beyond 300.

Wilken and Shen report that the coverage of sequence errors is less than $1 - 1/(l_{max} + 1)$ [25]. For $l_{max} = 10$, the sequence error coverage is less than 99.17%. To improve the error coverage, the intermediate signatures must be randomized [22, 25]. To do this in our signature model, random initial signatures are added after each reference signature. For the optimal case, on average the reference signatures account for 43% of the memory overhead and 45% of the performance overhead. Therefore, randomizing the signatures will increase the optimal performance overhead from 4.02% to 5.83% and the optimal memory overhead from 10.93% to 15.63%. The error coverage with randomized intermediate signatures is approximately $1 - 2^{-w} = 99.99 + \%$ for $w = 16$ [11, 25].

For the same performance and memory overhead, Saxena's Extended Precision Checksums can be used [16]. Extended Precision Checksums detect all single bit errors and all unidirected errors. In addition, the sequence error coverage approaches one as the number of sequence errors increases and the average detection latency is usually less than $l_{max}/2$.

We were not able to study the effect of interrupts and context switches on the error coverage. However, since the signature is disabled on a return from interrupt until the first reference signature, the error coverage will decrease as l_{max} increases.

7 Conclusions

In this paper we presented a signature insertion scheme with simple implementation complexity and low performance overhead. Our justifying arc insertion method has $O(N^2)$ algorithm complexity compared to the exponential complexity of previous node insertion methods. Furthermore, we proved that optimizing compiler techniques can be used to minimize the performance overhead for arc insertion and empirically proved that this optimized arc insertion minimizes the performance overhead due to justifying signatures.

We also performed experiments bounding the error detection latency and discovered that there is an inverse exponential relationship between the performance and memory overheads and the error detection latency. Using the MC68000 as our target processor, the performance and memory overheads for our benchmark set are relatively constant for detection latencies greater than 200 instruction cycles. For latencies between 10 and 200 cycles, the performance overhead ranges from approximately 15.76% to 4.02%. Likewise, the memory overhead drops from approximately 17.35% to 10.93%.

Acknowledgements

The authors would like to thank Michael Loui, Pohua Chang, John Fu, Paul Chen, Bob Dimpsey, and all members of the IMPACT research group for their support, comments and suggestions. The

authors would also like to acknowledge the contributions of Tom Conte for the use of his profiling package. This research has been supported by the Office of Naval Research under Contract N00014-88-K-0656, the National Science Foundation (NSF) under Grant MIP-8809478, a donation from NCR, and the National Aeronautics and Space Administration (NASA) under Contract NASA NAG 1-613 in cooperation with the Illinois Computer laboratory for Aerospace Systems and Software (ICLASS).

References

[1] A.V. Aho, R. Sethi, J.D. Ullman, *Compilers, Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley, 1986.

[2] W.C. Carter, "Improved Parallel Signature Checkers/Analyzers," *FTCS-16*, pp. 416-421, 1986.

[3] W. Cramer, G. Kane, *68000 Microprocessor Handbook*, Berkeley, CA: McGraw-Hill, 1986.

[4] X. Delord, R. Leveugle, G. Saucier, "Extended Duplex Fault Tolerant System with Integrated Control Flow Checking," *International Workshop on Defect and Fault Tolerance in VLSI Systems*, pp 98-109, 1989.

[5] J.B. Eifert, J.P. Shen, "Processor Monitoring Using Asynchronous Signatured Instruction Streams," *FTCS-14*, pp. 394-399, 1984.

[6] P.J. Fleming, J.J. Wallace, "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results," Computing Practices, Vol. 29, No. 3, pp. 218-221, March 1986.

[7] U. Gunneflo, J. Karlsson, J. Torin, "Evaluation of Error Detection Schemes Using Fault Injection by Heavy-Ion Radiation," *FTCS-19*, pp. 340-347, 1989.

[8] J.K.F. Lee, A.J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, pp. 6-22, January 1984.

[9] R. Leveugle, T. Michel, G. Saucier, "Design of Microprocessors with Built-In On-Line Test," *FTCS-20*, pp. 450-456, 1990

[10] D.J. Lu, "Watchdog Processors and Structural Integrity Checking," *IEEE Trans. on Computers*, Vol. 31, No. 7, pp. 681-685, July 1982.

[11] A. Mahmood, E.J. McCluskey, "Watchdog Processors: Error Coverage and Overhead," *FTCS-15*, pp. 214-219, 1985.

[12] A. Mahmood, E.J. McCluskey, "Concurrent Error Detection Using Watchdog Processors–A Survey," *IEEE Trans. on Computers*, Vol. 37, No. 2, pp. 160-174, February 1988.

[13] S. McFarling, J. Hennessy, "Reducing the Cost of Branches," *Proc. 13th Annu. Symp. on Comput. Arch.*, pp. 396-403, 1986.

[14] M. Namjoo, "Techniques for Concurrent Testing of VLSI Processor Operation," *Int. Test Conf.*, pp. 461-468, 1982.

[15] M. Namjoo, "Cerebus-16: An Architecture for a General Purpose Watchdog Processor," *FTCS-13*, pp. 216-219, 1983.

[16] N.R. Saxena, E.J. McCluskey, "Control-Flow Checking Using Watchdog Assists and Extended Precision Checksums," *FTCS-19*, pp. 428-435, 1989.

[17] M.A. Schuette, J.P. Shen, D.P. Siewiorek, Y.X. Zhu, "Experimental Evaluation of Two Concurrent Error Detection Schemes," *FTCS-16*, pp. 138-143, 1986.

[18] M.A. Schuette, J.P. Shen, "Processor Control Flow Monitoring Using Signatured Instruction Streams," *IEEE Trans. on Computers*, Vol. 36, No. 3, pp. 264-276, March 1987.

[19] J.P. Shen, M.A. Schuette, "On-Line Self-Monitoring Using Signatured Instruction Streams," *Int. Test Conf.*, pp. 275-282, 1983.

[20] J. Sosnowski, "Detection of Control Flow Errors Using Signature and Checking Instructions," *Int. Test Conf.*, pp. 81-88, 1988.

[21] T. Sridhar, S. Thatte, "Concurrent Checking of Program Flow in VLSI Processors," *Int. Test Conf.*, pp. 191-199, 1982.

[22] C. Tung, J. Robinson, "On Concurrently Testable Microprogrammed Control Units," *Int. Test Conf.*, pp. 895-900, 1986.

[23] N.J. Warter, W.W. Hwu, "A Software Based Approach to Achieving Optimal Performance for Signature Control Flow Checking," *FTCS-20*, pp. 442-449, 1990.

[24] N.J. Warter, W.W. Hwu, "Compiler-Assisted Signature Monitoring," Tech. Report, Center for Reliable and High-Performance Computing, University of Illinois, Urbana-Champaign, IL, (In preparation).

[25] K. Wilken, J.P. Shen, "Embedded Signature Monitoring: Analysis and Technique," *Int. Test Conf.*, pp. 324-333, 1987.

[26] K. Wilken, J.P. Shen, "Continuous Signature Monitoring: Efficient Concurrent-Detection of Processor Control Errors," *Int. Test Conf.*, pp. 914-925, 1988.