Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection

Z. Alkhalifa, V.S.S. Nair, *Member, IEEE,*
N. Krishnamurthy, and J.A. Abraham, *Fellow, IEEE*

Abstract—This paper evaluates the concurrent error detection capabilities of system-level checks, using fault and error injection. The checks comprise application and system level mechanisms to detect control flow errors. We propose *Enhanced Control-Flow Checking Using Assertions* (ECCA). In ECCA, branch-free intervals (BFI) in a given high or intermediate level program are identified and the entry and exit points of the intervals are determined. BFIs are then grouped into blocks, the size of which is determined through a performance/overhead analysis. The blocks are then fortified with preinserted assertions. For the high level ECCA, we describe an implementation of ECCA through a preprocessor that will automatically insert the necessary assertions into the program. Then, we describe the intermediate implementation possible through modifications made on gcc to make it ECCA capable. The fault detection capabilities of the checks are evaluated both analytically and experimentally. Fault injection experiments are conducted using FERRARI [1] to determine the fault coverage of the proposed techniques.

Index Terms—Control flow checking, assertions, fault injection, coverage, latency.

1 Introduction

Real time systems that are used in high dependability and integrity applications need to be designed with the capability to on-line detect and recover from the errors caused by hardware and software faults. Since the majority of errors are usually transient and not reproducible [2], off-line testing will not reliably detect them. Thus, it is imperative that the systems be designed with built-in concurrent error detection and recovery mechanisms. This paper presents a set of system-level checks comprising enhanced control flow checking using assertions (ECCA) at its high and intermediate levels for the concurrent detection of errors caused by hardware faults.

Various control-flow checking techniques have been proposed in the past to detect processor faults [3], [4], [5], [6], [7], [8], [9]. The techniques employ a watchdog processor to compute run-time signatures from the instructions and compare them with the precomputed signatures. These techniques need either additional hardware or modification of the existing hardware and are invariably nonportable to various platforms. Further, the current processor designs with built-in caches deny access to the instructions executed by the processor from the watch-dog processors, which effectively paralyzes the approaches. Complexity of modern compilers is yet another source of additional control-flow faults which cannot be handled by the existing methods as they assume error free object output from the compilers.

To circumvent these limitations, we have developed a high-level control-flow checking approach using assertions (CCA) [10], [11], [14]. In CCA, branch-free intervals in a given high-level language program are identified and the entry and exit points of the intervals fortified through preinserted assertions. The CCA approach is portable across architectures and requires no special hardware or database lookups to implement. It is implementable through a preprocessor based on the syntactic structure of the language and does not require generation and analysis of various paths in the program control-flow graph. CCA will detect hardware or compiler induced control flow faults. In this paper, we present an enhanced version of CCA targeted for real-time distributed systems (ECCA) for the detection of control-flow errors. Low overhead and low detection latency requirements had a major impact on the design of ECCA. Furthermore, ECCA inherits CCA's architectural portability which allows its implementation to be carried out on heterogeneous distributed systems. At ECCA's intermediate level implementation, programming language portability is also achieved.

Due to its robust nature, ECCA could easily be integrated with application specific data checks to complement its error detection capabilities [12], [13]. The data value checks are designed based on the assumption that hardware and software faults will eventually corrupt the data and produce wrong results. Data value checks can potentially provide coverage in the following situations where ECCA, or any other control-flow checking mechanism for that matter, will fail: 1) illegal branches within the branch free intervals causing data value errors, 2) incorrect decisions on conditional branches due to errors in the evaluation of the conditions, and 3) erroneous computed results, despite no control-flow faults.

The control-flow error detection capabilities of ECCA can be determined theoretically. However, the actual fault

- *Z. Alkhalifa and V.S.S. Nair are with the Computer Science and Engineering Department, Southern Methodist University, Dallas, TX 75275-0122. E-mail: zeyad@seas.smu.edu.*
- *N. Krishnamurthy is with the PowerPC Design Center, Somerset, Motorola, Austin, Texas.*
- *J.A. Abraham is with the Computer Engineering Research Center, The University of Texas at Austin, Austin, Texas.*

For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 109044.

1045-9219/99/$10.00 © 1999 IEEE

coverage provided by these checks in a real system cannot be assessed by a theoretical analysis as one cannot predict which of the faults in the system will translate into control-flow errors and data value errors. In this paper, we study the fault coverage provided by the technique with the help of fault injection experiments using FERRARI [1].

In Section 2, we describe ECCA. For completeness of discussion, we briefly describe CCA which was the precursor to ECCA. The experimental evaluation is given in Section 3 with a detailed discussion of various experimental results. Finally, we conclude the paper in Section 4 with some pointers towards future research.

1.1 CCA

In CCA, a program is split into a set of Branch Free Intervals (BFIs) and two identifiers are assigned to each of them. The Branch free interval IDentifier (BID) represents the branch free interval and is unique for every BFI. The Control Flow IDentifier (CFID) represents permissible control flow and is the same for all branch free intervals sharing a previous (parent) branch free interval.

The control flow checking is accomplished by setting and verifying the BIDs and CFIDs. At entry to a branch-free interval the current BID is assigned to the BID variable. Upon exit of the branch free interval, the current BID is verified. If a branch-free interval is entered from the middle, then the current BID will not match and an error will be detected.

The CFID is used to ensure that branch free intervals are executed in the correct order. The CFID's are stored in a two element queue. The queue is initialized to contain the CFID of the first branch free interval. Upon entry to a branch-free interval, the CFID of the next branch free interval is placed on the queue. Upon exit of a branch free interval, a CFID is dequeued, and verified to match the current CFID. The enqueue and dequeue operations must first verify that the queue is in the correct state for the operation and then perform the actual enqueue or dequeue. That is, the enqueue must fail if the queue is full and the dequeue must verify that the next element is the correct element. Early verification of the queue status reduces the average fault detection latency.

Fig. 1 shows the control flow graph of a simple IF_THEN_ELSE construct, while Fig. 2 demonstrates the procedure of assigning BIDs and CFIDs to branch-free intervals for the same construct. Also shown in Fig. 2 are the contents of the two element queue and the BID variable at a given point of execution. Note that since BFI B and BFI C share a common parent, they are assigned the same CFID. A control flow error is detected if any one of the following situations occur:

1. The current BFI's BID does not match the BID variable. Hence, an illegal branch has occurred from the previous BFI to the middle of the current BFI.
2. An overflow in the two element queue—this situation arises if an illegal branch happens from the middle of the previous BFI to the beginning of the current BFI.
3. Underflow in the two element queue—happens when an illegal branch is taken from the end of the previous BFI to the middle of the current BFI.
4. Dequeue mismatch—means the branch is to a nonpermissible BFI.
5. BID mismatch—this occurs when an illegal branch is performed from the middle of the previous BFI to the middle of the current BFI.

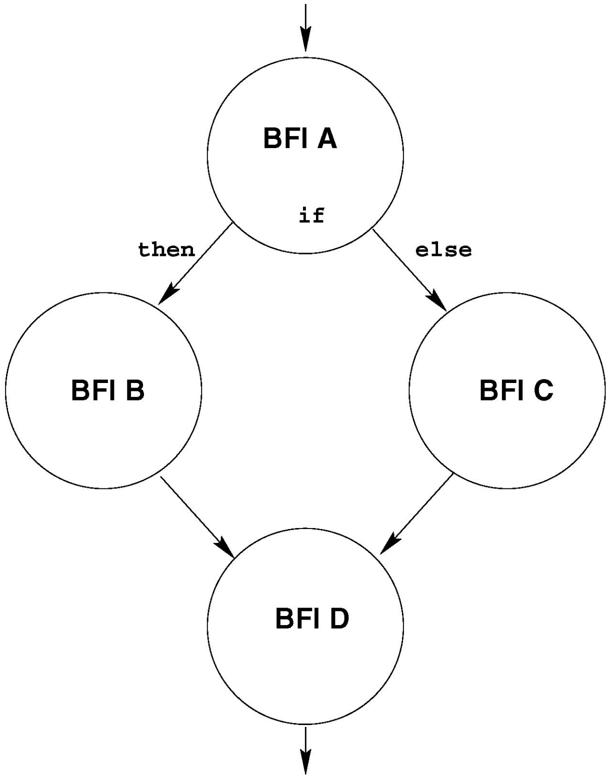

Fig. 1. Control flow graph of the IF_THEN_ELSE construct prior to the placement of CCA assertions.

CCA can detect all the single control-flow faults and most of the multiple control-flow faults. However, it comes with a price, the high overhead. In situations where BFIs are small, adding multiple lines of code to enqueue, dequeue, examining the queue, setting a variable, and testing the variable could prove inefficient. Also, it could increase the probability of a control flow error occurring due to the increased size of the program. Moreover, the added code to the original program, at a minimum, consumes three memory locations (one for the BID variable and two for the two element queue) which might translate into three registers, hence leaving fewer registers for the original program to use. Furthermore, the assertions themselves contain uncovered branches. Therefore, a control flow error in the assertion code could report a fault when no fault had occurred in the original program. Finally, there does exist a remote case when CCA fails to detect a control flow error when one actually occurred. This case happens when two or more parent BFIs have at least a common child. The CFID of all the children of these parents would be equal, as shown in the example of Fig. 3. In the example, an illegal branch from the end of BFI B to the beginning of BFI C, or an illegal

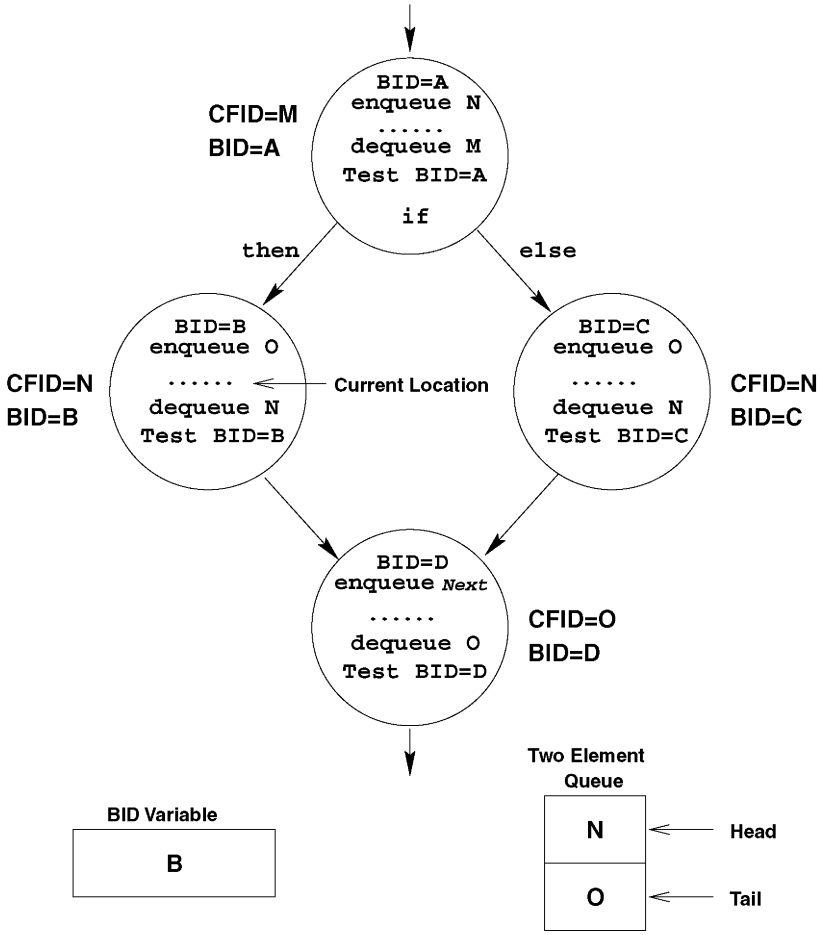

Fig. 2. Assignment and checking of IDs for IF_THEN_ELSE construct after the placement of CCA assertions.

branch from the end of BFI A to the beginning of BFI F, will not be detected due to the equal CFID in BFIs C, D, and F.

2 Enhanced Control-Flow Checking Using Assertions (ECCA)

Enhanced control-flow checking using assertions (ECCA) is the successor to control-flow checking using assertions (CCA) which has been proposed in the past [10], [11], [14] for the concurrent detection of control-flow errors. In this section, we present ECCA in detail and show how the limitations of CCA are overcome by the new technique.

ECCA is proposed to address the aforementioned limitations of CCA. It is implemented at both the high and intermediate levels. Our choice of an intermediate level language was the *Register Transfer Language* (RTL) used in GNU's compilers. Through the addition of an exception handler, ECCA adds only two instructions per block (a collection of consecutive BFIs), when implemented for the high level language. Neither of these assertions contain a branch and only one additional variable is needed. However, its low level representation (machine code) will contain two branches; therefore, we propose a variation of the assertions when implementing it at the intermediate level. This method will detect hardware or compiler induced control flow faults. The low overhead and low detection latency make ECCA ideal for real-time systems. Moreover, its architectural portability allows its implementation to be carried out on heterogeneous distributed systems.

2.1 High Level ECCA

In this section, we demonstrate the method and implementation of ECCA at the high level code. Due to its widespread use and familiarity, we chose C as our language of implementation.

2.1.1 Assertions in High Level ECCA

Unlike CCA, ECCA divides the program into a set of *blocks*. We define a *block* as a collection of consecutive BFIs in which there exists a single entry point and a single exit point. An exit point could only branch to an entry point of a block. For maximal coverage, a block will consist of only one BFI; however, this will cause maximum overhead since each BFI will contain two assertions.

A unique prime number ID larger than 2 called Block IDentifier (also BID) is assigned to each block. Thereafter, two lines of code are asserted into each block. The first is a simple assignment executed upon entry of a block and is of the following form:

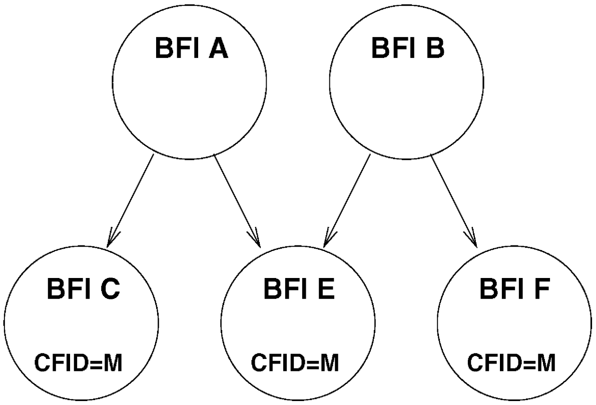

Fig. 3. The control flow graph that could create an undetected control flow error in CCA.

$$id \leftarrow \frac{BID}{\overline{(id \bmod BID)} \cdot (id \bmod 2)},$$

where the elements in the assignment are defined as follows:

id: A global integer variable which is updated during execution time upon entry into and exit from each block.

BID: *Block IDentifier* is a unique prime number larger than 2 generated for each block during preprocess time (hard coded into the program).

Note that the assignment will result in a divide by zero error if either $\overline{id \bmod BID} = 0$ or $id \bmod 2 = 0$.

The second assertion is also an assignment. It is executed immediately before departing the block. Therefore, it is placed at the end of the block. The assignment is as follows:

$$id \leftarrow NEXT + \overline{\overline{(id - BID)}},$$

where *NEXT* is an integer number generated at preprocess time and is equal to the product of permissible blocks' BIDs from the current block. Therefore, $NEXT = \Pi BID_{Permissible}$. Note that, due to the double negation on (*id-BID*), the result of $\overline{\overline{(id - BID)}}$ is either a 0 or 1. For convenience, in the subsequent discussions, we call the first assertion the SET assertion and the second one the $TEST$ assertion.

In practice, a preprocessed block using the C programming language would have the structure depicted in the following:

```
/*Beginning of the block*/
id = <BID>/((!(id\%<BID>))*(id\%2));
          . . . block BODY . . .
id = <NEXT>+!!(id-<BID>);
/*End of the block*/
```

Remember that <NEXT> and <BID> represent integer numbers generated at preprocess time, while *id* is a variable updated at run time.

As an example, consider the following code which needs to be preprocessed using the ECCA. For simplicity we assume that a block contains only one BFI.

```
. . .
/*Beginning of original code*/
foo=1;
if foo
{boo=10}

else
{hoo=20}
foo=0;
/*End of original code*/
```

Preprocessing will modify the code to insert assertions as follows:

```
/*Beginning of preprocessed code*/
. . .
/* <BID> is 3 */
foo=1;
id = 35+!!(id-3);
/*Note that the above 35 is a result of
multiplying permissible blocks
BIDs, i.e., 5*7 */
if foo
{
/* <BID> is 5 */
  id = 5/((!(id%5))*(id%2));
  boo=10;
  id = 11+!!(id-5);
}
else
{
/* <BID> is 7 */
  id = 7/((!(id%7))*(id%2));
  hoo=20;
  id = 11+!!(id-7);
}
/* <BID> is 11*/
id = 11/((!(id%11))*(id%2));
. . .
```

To further illustrate the working of the ECCA method, we show the preprocessing of a while loop nested within an if statement. We also assume one BFI per block.

```
. . .
/*Beginning of original code*/
... BLOCK BODY 1 ...
if foo
{
  ... BLOCK BODY 2 ...
  while foo
  {
    ... BLOCK BODY 3 ...
  }
  ... BLOCK BODY 4 ...
}
else
{
  ... BLOCK BODY 5 ...
}
... BLOCK BODY 6 ...
/*End of original code*/
. . .
```

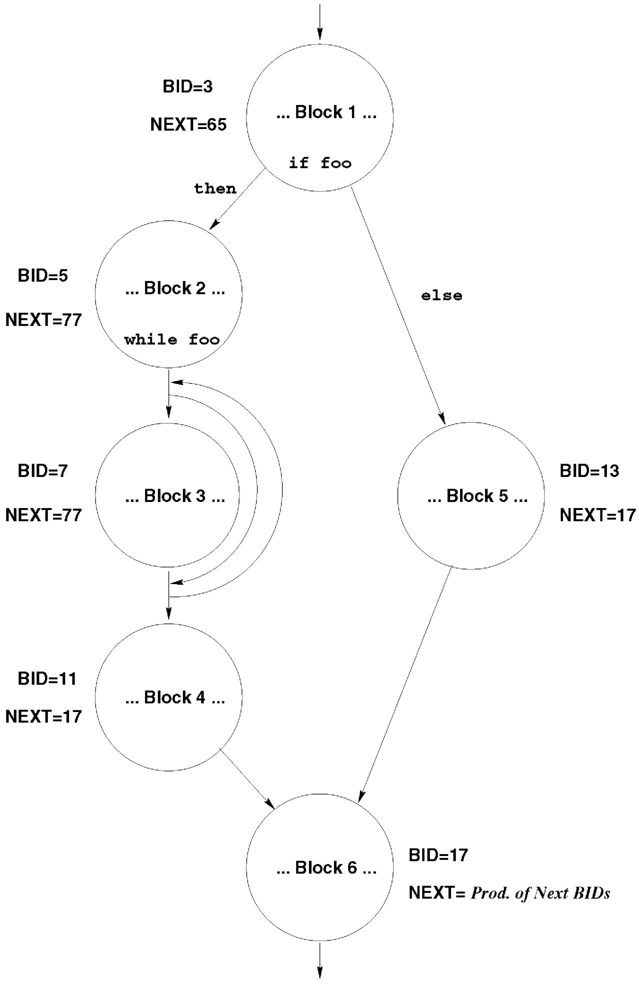

Fig. 4. While nested in If-Then.

The ECCA preprocessor will first split the program into blocks (remember for this example the block consists of a single BFI) and then generate a BID and NEXT value for each block, as demonstrated in Fig. 4.

Upon careful examination of Fig. 4, one notices that the value of NEXT in block 2 is 77. Therefore, it is possible to branch from block 2 to block 4 bypassing block 3 (condition of the while loop fails before performing the first iteration of the loop). In block 3, the value of NEXT is also 77. This is because block 3 could either branch to itself or to block 4.

The preprocessed code is as follows:

```
. . .
/*Beginning of preprocessed code*/
... BLOCK BODY 1 ...
id = 65+!!(id-3);
if foo
{
   id = 5/((!(id%5))*(id%2));
   ... BLOCK BODY 2 ...
   id = 77+!!(id-5);
   while foo
   {
     id = 7/((!(id%7))*(id%2));
     ... BLOCK BODY 3 ...
     id = 77+!!(id-7);
   }
   id = 11/((!(id%11))*(id%2));
   ... BLOCK BODY 4 ...
   id = 17+!!(id-11);
}
else
{
   id = 13/((!(id%13))*(id%2));
   ... BLOCK BODY 5 ...
   id = 17+!!(id-13);
}
id = 13/((!(id%13))*(id%2));
... BLOCK BODY 6 ...
id = Prod. of Next BIDs +!!(id-5);
/*End of preprocessed code*/
. . .
```

Until now, all of our examples dealt with blocks containing one BFI. Though effective in coverage, doing so gives a high overhead since each BFI will contain two added lines of code. Moreover, if a loop exists, the asserted code will be executed numerous times. Therefore, grouping a number of consecutive BFIs should significantly reduce overhead; however, detection latency will be increased [19]. As an example, consider the *while* nested inside an *if-then* depicted in Fig. 4. If we merge blocks 2, 3, and 4, we not only reduce the number of assertions added, but also avoid executing the assertions in every iteration of the while loop. This greatly reduces the overhead without sacrificing much coverage. Fig. 5 demonstrates the new representation of the while nested in If-Then.

Consider a program of size l blocks. Furthermore, n of the l blocks will actually be executed. Each block contains m BFIs, and the execution time of each BFI is w units. We also give the following conservative assumptions:

- The overhead of the two assertions is equivalent to w (this is a worst case assumption).
- In the event of an illegal branch, the probability of branching to any point in the program is equal.

The program execution time prior to placing assertion is

$$T_{prior} = n \cdot m \cdot w,$$

and the execution time after adding the assertions will be

$$T_{after} = n \cdot (m + 1) \cdot w$$

ECCA has the ability to detect all single control flow errors crossing block boundaries (we will show that later). Therefore, in the event of a control flow error, the probability of not crossing a block boundary is

$$P_{undetect} = 1/l,$$

hence, the coverage is

$$Coverage = (1 - P_{undetect}) \cdot 100 = (1 - 1/l) \cdot 100,$$

Note that we relate *coverage* only to the detection of control flow errors.

To demonstrate the effectiveness of grouping BFIs in blocks, consider a moderate size program containing 10,000

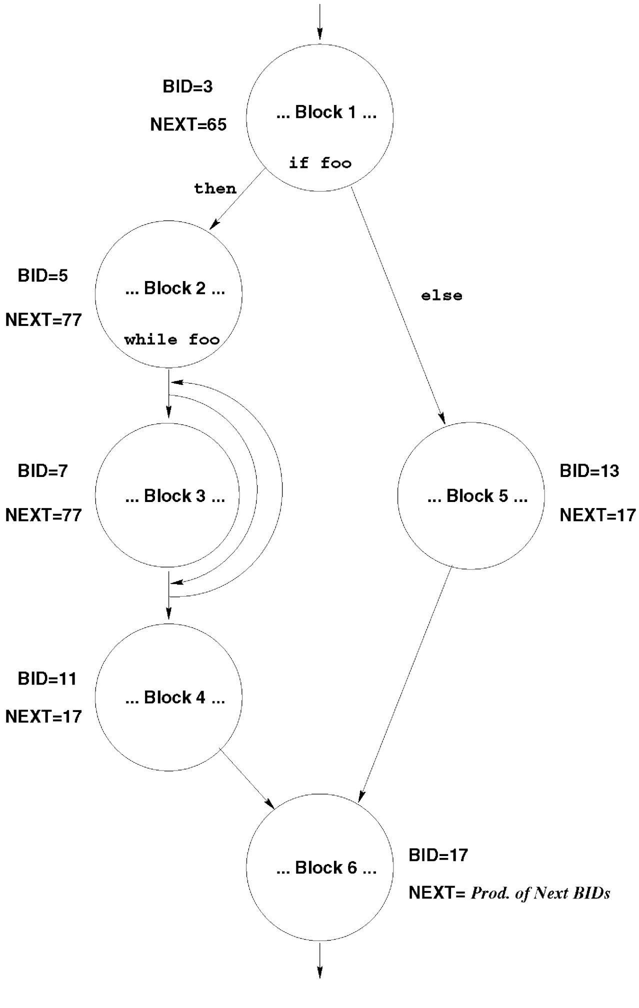

Fig. 5. Merging three BFIs into a block.

BFIs where we group every 20 BFIs into a block. Therefore, $l = 500$ and $m = 20$. If we consider that only 10 percent of the code will ever be executed, then $n = 50$ blocks. The execution time prior to adding assertions is

$$T_{prior} = 50 \cdot 20 \cdot w = 1,000w,$$

and the execution time after adding the assertions is

$$T_{after} = 50 \cdot (20 + 1) \cdot w = 1,050w.$$

Therefore, the percentage of overhead due to the assertions is

$$\frac{(T_{after} - T_{prior}) \cdot 100}{T_{prior}} = 5\%.$$

The coverage would be

$$Coverage = (1 - 1/l) \cdot 100 = (1 - 1/500) \cdot 100 = 99.8.$$

Thus, we obtain 99.8 percent coverage by adding only 5 percent overhead.

The above is merely an example of the power of ECCA. It may not resemble the actual overhead. We predict the overhead should be lower as we stated in our assumptions. The decision of block size and choosing the first and last BFI of a block, plays a major role in overhead reduction. The following are a few guidelines to aid in overhead reduction:

- Choose an appropriate block size compared to the size of the program. A larger block means less overhead, but it also means less coverage and increased detection latency.
- Always try to start a block with a nonloop BFI. Doing so prevents from executing the assertions in every iteration of the loop.
- The start of a function means the start of a block. If the function is too small, attempt to inline it at compile time (this is only possible in intermediate level ECCA).

2.1.2 High Level ECCA Preprocessor Implementation

The ECCA preprocessor consists of three major components: a pseudo control-flow C-parser, a lexical analyzer, and a translation unit located between the parser and the lexical analyzer which filters out unneeded C code that does not cause a begin or end of a BFI.

Since the parser needs to trace only the statements that influence the control flow of the program, a filter is used to reduce standard C statements to pseudostatements. Statements that do not change the control flow are reduced to null statements. As an example, see the following code:

```
while (x < y) {
x++;
y–;
}
```

is reduced to

```
while {
;
;
}
```

Note that since the increment and decrement operations (performed on the x and y variables) do not affect the format or location of the assertions, they were reduced to semicolons by the filtering unit of the preprocessor. By doing so, we not only increase the efficiency of the preprocessor but also eliminate unneeded parsing that would be handled in the compilation phase of the program to be preprocessed. Furthermore, such an implementation increases the portability of the preprocessor giving us the ability to easily modify it to recognize the control flow of programs coded in other programming languages. The following grammar was used for the parser:

$$
\begin{aligned}
&program \rightarrow function \\
&\qquad\qquad \mid program\ function \\
&function \rightarrow \text{ID}\{statement\_list\} \\
&statement\_list \rightarrow statement \\
&\qquad\qquad\quad \mid statement\_liststatement \\
&\qquad\qquad\quad \mid \epsilon \\
&statement \rightarrow compound\_statement \\
&\qquad\qquad \mid selection\_statement \\
&\qquad\qquad \mid iteration\_statement \\
&\qquad\qquad \mid jump\_statement
\end{aligned}
$$

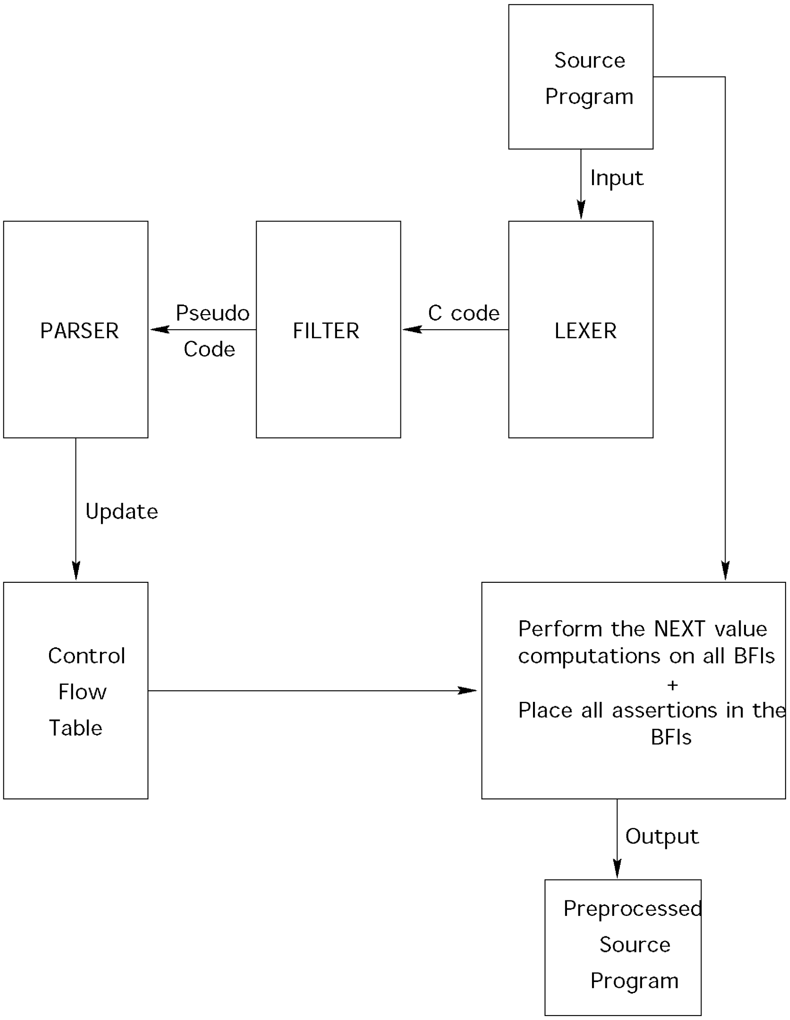
Source Program
Input
PARSER
Pseudo Code
FILTER
C code
LEXER
Update
Control Flow Table
Perform the NEXT value computations on all BFIs
+
Place all assertions in the BFIs
Output
Preprocessed Source Program

Fig. 6. ECCA preprocessor design.

```
                | labeled_statement
                | ;
compound_statement → {statement_list}
selection_statement → IF statement_list
                      | IF statement_list ELSE statement_list
                      | SWITCH statement_list
iteration_statement → WHILE statement_list
                      | DO statement_list WHILE
                      | FOR statement_list
jump_statement → BREAK
                 | CONTINUE
                 | GOTO ID
labeled_statement → ID:
                    | CASE ID
                    | DEFAULT
```

A table is used to store the needed information about the control flow of the program. Each BFI is represented by a row in the table. The row consists of the following four fields: the BID assigned value, the NEXT computed value, the GENERAL assigned value (if needed), and a linked list of pointers to the indices of the rows in the table representing the permissible BFIs from the current BFI.

Upon encountering a statement that affects the control flow of the program, the parser updates the linked list. When the parsing stage is complete, a unique prime number is placed in every BID cell in the table. Furthermore, the GENERAL value is entered for the BFIs affected by the switch or case statements. The NEXT field is then computed by simply traversing the list and multiplying the corresponding BID cells of the indices contained in it. The source input program is then read once again, converting every single statement within a control statement to a compound statement. The formulation of the *id* assignment statements is done through referencing the table. Finally, the id assignments are asserted in the source. Fig. 6 sketches the ECCA's preprocessor design.

2.2 Intermediate Level ECCA

High level code assertions guarantee architectural portability; however, they fall short in high level programming language portability. Moreover, compiler optimizations might influence the added assertions. In this section, we demonstrate how ECCA could be implemented at the intermediate level code thus guaranteeing both language and architecture portability [15]. We

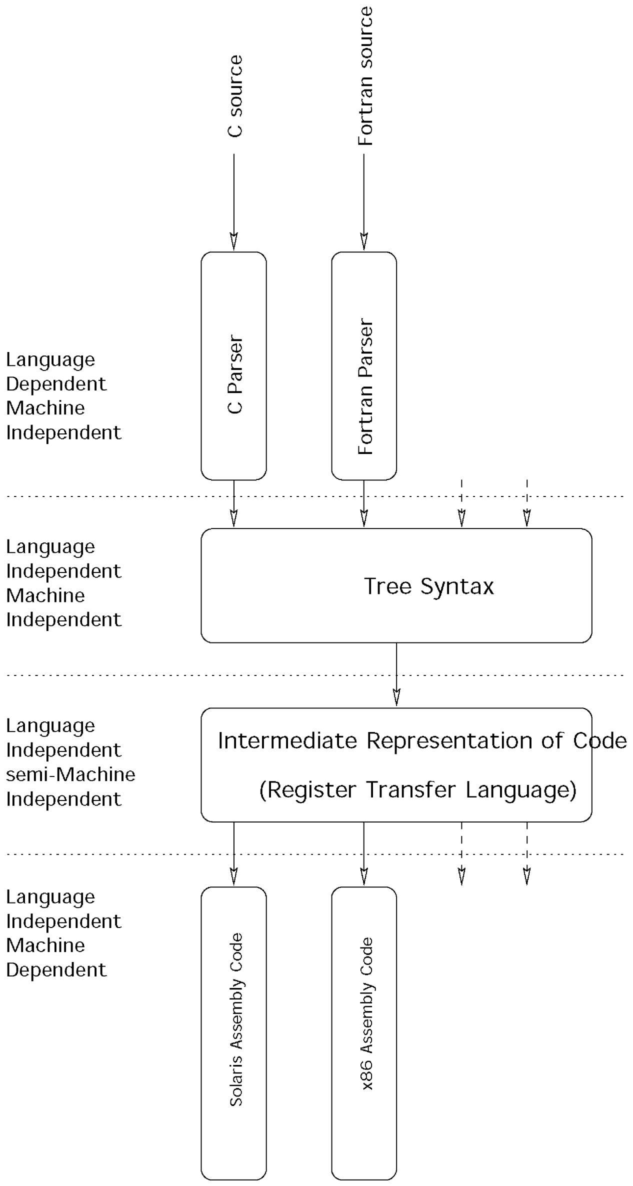
C source
Fortran source
Language Dependent Machine Independent
C Parser
Fortran Parser
Language Independent Machine Independent
Tree Syntax
Language Independent semi-Machine Independent
Intermediate Representation of Code
(Register Transfer Language)
Language Independent Machine Dependent
Solaris Assembly Code
x86 Assembly Code

Fig. 7. Overview of gcc's compilation stages.

implement ECCA at the RTL stage which is GNU's representation of an intermediate level language used for its compilers. Keep in mind that RTL is only GNU's implementation, hence, language and architecture portability of intermediate level ECCA is only good for GNU's compilers. Other compilers might have different intermediate level representations. Nevertheless, porting ECCA to other compilers could easily be done (given the availability of the source code).

GNU offers a variety of free compilers. The source code for the compilers is also available for free. The programming languages include C, C++, Ada, Fortran, Pascal, and many others. Moreover, it is portable across many platforms. One could easily perform cross platform compilation. In order to support many languages for many platforms, the compilation process is done through several stages in which the source code is transformed from one form into another. All languages for all platforms meet at the syntax tree and RTL levels. Most of the compiler's work is done at the RTL level [18]. This organization makes it easy to support new languages or new platforms since it only entails changing the front or back end of the compiler. Fig. 7 gives an overview of Gnu's compilation stages.

Even though RTL is somewhat dependent on the underlying platform, our implementation is fully platform independent. In order to understand how we achieve this independence, consider Fig. 8. The RTL stage of compilation is divided into numerous passes. Each pass performs a certain optimization or function on the compiled program. The deeper the code goes down in the sequence of passes, the more machine dependent it becomes. We now introduce a new pass named the ECCA pass. This is the pass where all of the analysis and tasks done by ECCA will be carried out. In choosing the optimal location for the ECCA pass, we considered going as low as possible, thus bypassing most machine independent optimizations. A satisfactory location is between the *second common subexpression elimination pass* and the *stupid register allocation pass*. We also achieve independence by using the compiler's internal routines for code flow analysis and assertion code generation in the ECCA pass. By implementing ECCA at the RTL level of compilation we achieve the following:

- Portability across programming languages.
- Reduction in overhead since the placement of the assertions is performed after most of the compiler's optimizations are done.
- More control over system resources needed for the assertions.

2.2.1 Assertion in Intermediate Level ECCA

In RTL, the maximum number of destinations that an instruction containing a branch could have is two. All loops and If-Then-Else instructions are represented by simple conditionals and jump statements. The only condition where the maximum number of destinations is not two is in the case statement (switch in C). However, a mandatory grouping of the case into a block would resolve that. To reduce the overhead, we modify the SET and TEST assertions to take advantage of this property. Each assertion will now contain multiple RTL instructions (also called *insns*). Nevertheless, we will show that a control flow error from or to the middle of an assertion will be detected.

The SET assertion placed at the entry point of a block should be

$$r_1 \leftarrow (r_1 - BID) \cdot (r_2 - BID),$$

$$r_1 \leftarrow \frac{BID - 1}{\left(\frac{r_1+1}{r_1 \cdot 2+1}\right)},$$

where r_1 and r_2 are global registers and BID is a unique prime number greater than 2 generated for each block at compile time.

Under correct execution, either r_1 or r_2 at the beginning of the SET assertion will contain the current BID value. This

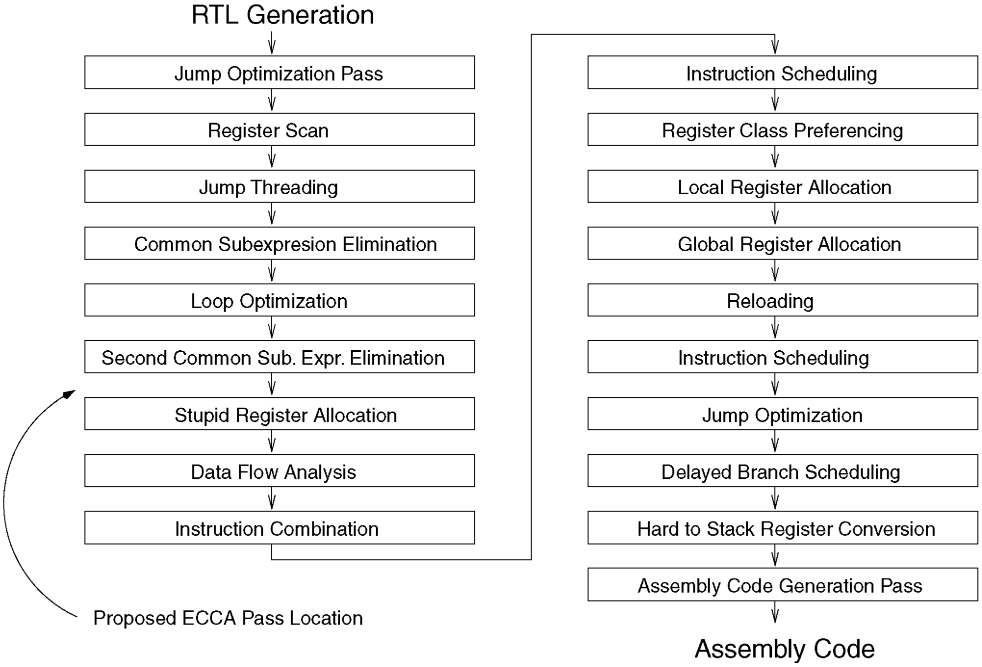
RTL Generation
Jump Optimization Pass
Register Scan
Jump Threading
Common Subexpresion Elimination
Loop Optimization
Second Common Sub. Expr. Elimination
Stupid Register Allocation
Data Flow Analysis
Instruction Combination
Proposed ECCA Pass Location
Instruction Scheduling
Register Class Preferencing
Local Register Allocation
Global Register Allocation
Reloading
Instruction Scheduling
Jump Optimization
Delayed Branch Scheduling
Hard to Stack Register Conversion
Assembly Code Generation Pass
Assembly Code

Fig. 8. RTL passes.

value was set from the previous TEST assertion. After executing the first assignment of the SET assertion, r_1 should be zero if no control flow error has occurred and nonzero in the event of a control flow error. In the second assignment, r_1 will be assigned $\mathrm{BID}+1$ if r_1 is zero (no error). However, if r_1 is nonzero, a divide by zero exception will be raised from the assignment, thus indicating that a control flow error has occurred. The following pseudocode represents the intermediate representation of the SET assertion:

```
$tmp_1 \leftarrow BID + 1$
$tmp_2 \leftarrow r_1 - BID$
$tmp_3 \leftarrow r_2 - BID$
$r_2 \leftarrow tmp_2 \cdot tmp_3$
$tmp_2 \leftarrow r_2 + 1$
$tmp_3 \leftarrow$ shift $r_2$ left by one
$tmp_3 \leftarrow tmp_3 + 1$
$tmp_2 \leftarrow tmp_2 / tmp_3$
$r_1 \leftarrow tmp_1 / tmp_2$
```

Note that if a control flow error results in an illegal branch to anywhere in the middle of this assertion, r_1 will not be set to $\mathrm{BID}+1$, causing the error to be detected as we will show when we present the TEST assertion.

Even though the above intermediate representation of the SET assertion might seem lengthy, it is not time consuming if the program is executing correctly. The multiplication in the fourth instruction is a multiplication by zero and the two divisions are divisions by 1. Remember that the above is equivalent to the low level machine representation of code in terms of size.

The TEST assertion is placed at the exit point of the block. It sets r_1 and r_2 to the BIDs of the first and second permissible blocks, respectively. The following is the form of the assertions:

$$r_1 \leftarrow (r_1 - BID) \cdot NEXT_1$$
$$r_2 \leftarrow (r_1 - BID) \cdot NEXT_2,$$

where $NEXT_1$ and $NEXT_2$ represent the BIDs of the permissible blocks. Remember that there is only a maximum of two permissible blocks when implementing ECCA at the RTL level.

Under correct execution, $(r_1 - BID)$ will always equal 1, hence, r_1 and r_2 will be set to $NEXT_1$ and $NEXT_2$, respectively. In the event of a control flow error, $r_1 - BID$ will not equal to one. Therefore, r_1 and r_2 will be set to a nonprime multiple of the permissible blocks BID's. This in turn will cause the error to be detected at the next block's SET assertion.

The following pseudocode gives the intermediate representation of the TEST assertion:

```
$tmp_1 \leftarrow r_1 - BID$
$r_1 \leftarrow tmp_1 \cdot NEXT_1$
$r_2 \leftarrow tmp_1 \cdot NEXT_2$
```

Note that in the event of an illegal branch to the middle of the assertion tmp_1 will not be equal to one causing the error to be detected at the next block's SET assertion. Under correct execution, the two multiplications are multiplications by 1; therefore, overhead is kept minimal.

2.2.2 Error Detection Capabilities of ECCA

By partitioning the program into blocks, a control flow fault can be defined as a fault which incorrectly transfers control from one block to another. For an analytical evaluation of the detection capabilities of ECCA, we consider a fault model that includes control-flow errors of following types:

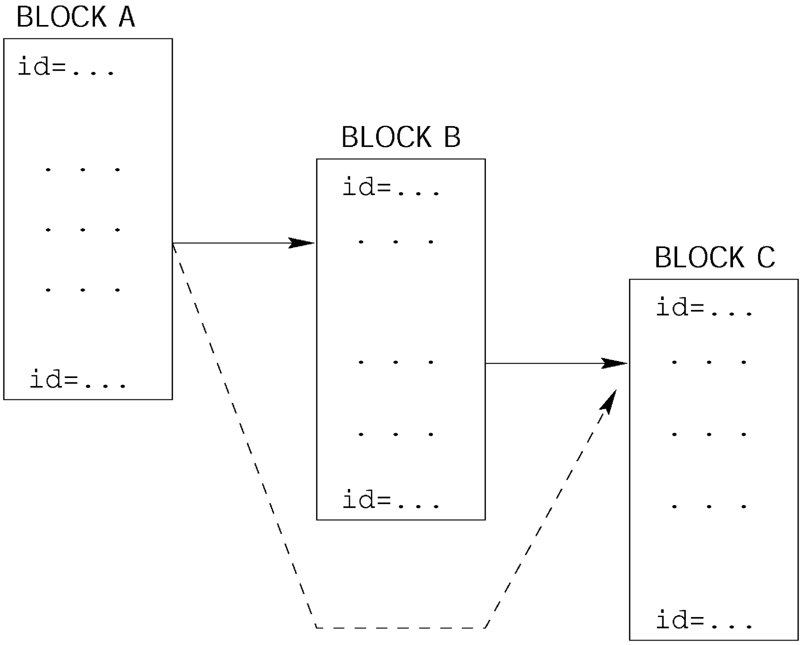

Fig. 9. Reduction of a multiple control flow error.

Type1: A fault causing a branch from the middle of a block to another.

Type2: A fault causing a branch to the middle of a block.

Type3: A fault causing a branch to an illegal block.

What could be considered as data value faults are not covered by this method. This includes illegal branches not crossing a block boundary and incorrect decisions on conditional branches. These faults could be detected using data value error detection mechanisms.

Theorem. *The ECCA method is capable of detecting all single control flow faults crossing block boundaries.*

Proof. We prove the theorem by showing that all the three types of control-flow faults will be detected by ECCA.

Type1: A fault causing a branch from the middle of a block to another.

Case 1: *From the middle of a block to the beginning of another.* In this case, the control flow has skipped the $TEST$ assertion in the previous block and, hence, the value of id upon entry to the new block is equal to the BID of the previous block. This will cause a division by zero exception during the execution of the SET assertion in the current block.

Case 2: *From the middle of a block to the middle of another.* In addition to skipping the $TEST$ assertion in the previous block, in this case, the control flow has skipped the SET assertion in the current block also. The value of *id* remains equal to the BID of the previous block until the control-flow reaches the $TEST$ assertion of the current block. Since $id \neq BID$, $\overline{(id - BID)} = 1$. This will cause *id* to be set to an even number by the $TEST$ assertion in the current block since $\Pi BID_{Permissible}$ is always odd. Now, at the beginning of the next block, its SET assertion will raise a divide by 0 exception as $(id \ mod \ 2) = 0$.

Case 3: *From the middle of a block to the end of another.* If the branch was to a location before the $TEST$ assertion, it falls under *Case 2*. Otherwise, it falls under *Case 1*.

Type2: A fault causing a branch to the middle of a branch-free interval.

Case 1: *From the beginning of a block to the middle of another.* Since the SET assertion in the current block is skipped, $id = \Pi BID_{permissible}$. Now, the $TEST$ assertion in the block will raise an exception as $id \neq BID$.

Case 2: *From the middle of a block to the middle of another.* This is the same as *Case 2* of *Type 1*.

Case 3: *From the end of a block to the middle of another.* Similar to *Case 1* of this type.

Type3: A fault causing a branch to an illegal block.

If the current block is not legitimate, the value of *id* (equal to $\Pi BID_{permissible}$), set by the $TEST$ assertion of the previous block is not divisible by BID. Therefore, the set assertion in the current block will give rise to an exception as $\overline{(id \bmod BID)} = 0$. □

2.2.3 Multiple Control Flow Errors

The ECCA method is designed to detect all single control flow errors. However, it is also capable of detecting most multiple control flow errors. We say *most* since there exists a single remote case in which a multiple control flow error will not be detected.

Given three blocks A, B, and C, a multiple control flow error can occur only if an illegal branch happens from the middle of block A to the middle of block B, and then from the middle of block B to the middle of block C, and so on. If, in the process of illegally branching from one block to another, we execute the SET or $TEST$ assertions of each block, it will be detected according to the theorem.

TABLE 1
Selected Transient Error Models

Model		Description
1	AddIF	address line error resulting in executing a different instruction
2	AddIF2	address line error resulting in executing two instructions
3	AddOF	address line error when a data operand is fetched
4	AddOS	address line error when an operand is stored
5	DataIF	data line error when an opcode is fetched
6	DataOF	data line error when an operand is loaded
7	DataOS	data line error when an operand is stored
8	CndCR	errors in condition code flags

Therefore, each illegal branch in the sequence has to happen from the middle of a block to the middle of another to be considered as an undetectable multiple control flow error. Since it involves only instructions not effecting the value of the *id* variable, it could be reduced to a single control flow error from the first block to the last. To clarify this, we consider the three blocks A, B, and C in Fig. 9. Note that the figure shows a control flow error from A to B to C is reduced to an error from A to C. The only situation that will cause a multiple control flow error to not be detected by ECCA is when the first and last blocks in the sequence of illegal branches are the same. Therefore, this case is reduced to a control flow error within a block.

2.2.4 Error Handling in ECCA

If the desire is to terminate upon encountering a control flow error, then what we have described thus far should suffice. However, if either knowing the cause of termination or recovering from the control flow error is considered, we need a mechanism to enable us to distinguish whether a divide by zero exception happened from the original code or the asserted code. In this section, we consider two mechanisms. The first mechanism uses simple if-then statements while the second mechanism inserts an exception handler.

An *if-then* statement could be used to transfer control to a control flow error handling routine. To that end, the assertion at the beginning of a block would be as follows:

```
/* Beginning of BLOCK*/
If !((!(id%<BID>))*(id%2))
    . . .Go to Handling Routine. . .
else
{
    id=<BID>;   . . .Body of BLOCK. . .
```

Note that ECCA at the intermediate level could be handled in a similar fashion.

Even though a simple *if-then* would work, it introduces a branch in the assertion, thus reducing coverage. Exception handling separates error-case from normal-case, therefore increasing performance. In other words, if something works fine 99.999 percent of the cases, then the normal-case code should be optimized for this. And the exception case should handle the .001 percent of error cases.

To install an exception handler we include the library signal.h and add a global routine called catchfpe().

```
/* Beginning of program*/
#include <signal.h>
. . .
#define SIGFPE 8
. . .
void catchfpe() {
        if id is prime then data divide by zero
        else control flow error divide by zero we then
                got handling routine
}
```

Note that if a control flow error caused the divide by zero exception, *id* will always be nonprime. As for the intermediate representation of ECCA, one could check r_1. If r_1 is an even number, then it is a data divide by zero; otherwise, it is a control flow error divide by zero.

3 Experimental Evaluation

The evaluation approach taken in this study involves the use of a software-based fault and error injection tool FERRARI [1]. FERRARI can inject both transient and permanent faults. The transient fault duration is one instruction cycle while the duration of a permanent fault might extend throughout the entire execution of the application.

Experiments presented were conducted on a SUN SPARC workstation to study the behavior of the target system when injected with faults and errors and to determine the effectiveness of the error detection and correction capabilities of ECCA.

For every experiment, we selected the fault type, the fault model to be injected (Table 1) and the number of fault/error injection runs. In each run, the bit position(s), the selected register to be faulted (if any), and the location (address inside the program code including libraries) at which a fault/error is injected were randomly selected. When the program execution reaches the location where the

TABLE 2
Error Detection Coverage for Test1 without ECCA

Error Models	Detection Coverage without ECCA				
	System Detection	Timeouts	User Detection	Undetected	Overall Coverage
AddIF	61.8%	0.6%	12.1%	25.6%	74.5%
AddIF2	82.5%	0.4%	6.3%	10.8%	89.2%
AddOF	71.2%	0.1%	7.7%	21%	79%
AddOS	57.8%	0%	6.7%	35.5%	64.5%
DataIF	61%	0.6%	14.5%	23.9%	76.1%
DataOF	17.7%	1.9%	12.5%	67.9%	32.1%
DataOS	31.3%	0.4%	9.0%	59.3%	40.7%
CndCR	0%	0%	7.7%	92.3%	7.7%

TABLE 3
Error Detection Coverage for Test1 with ECCA

Error Models	Detection Coverage with ECCA					
	System Detection	Timeouts	User Detection	ECCA	Undetected	Overall Coverage
AddIF	46.1%	0.1%	2.7%	44.7%	6.4%	93.6%
AddIF2	74.2%	0%	0.9%	22.6%	2.3%	97.7%
AddOF	63.6 %	0.1%	1.4%	30.2%	4.7%	95.3%
AddOS	65.3%	0%	1. 5%	24.7%	8.5%	91.5%
DataIF	49.1%	0.1%	2.7%	43.6%	4.6%	95.4%
DataOF	4.8%	0.5%	3.5%	72.9%	18.3%	81.7%
DataOS	4.9%	1.1%	1.8%	83.4%	8.8%	91.2%
CndCR	0%	0%	2%	76.5%	21.5%	78.5%

fault should be injected[1] the program execution is stopped and a software trap is activated. At this point, the error is injected by modifying the internal state of the processor with respect to the requested error model [1]. The program is then allowed to proceed.

Results for over 400,000 injected errors are presented in this paper. The number of runs for every experiment was based on obtaining the same system behavior $(+/-0.5\%)$ even after increasing the number of runs in the experiment. The system behavior is defined as the percentages of "detected" and "undetected" errors, as well as the contribution of each of the embedded error detection mechanisms. For some of the experiments, the response of the system became consistent at 10,000 runs, while for others, the behavior of the system became consistent at 20,000 runs.

In each experiment conducted, a selected application is first run without injecting an error in the system. The output, named *reference*, is written to a file for future comparisons. A fault or error is injected into the system while running the application. If no error detection mechanism is triggered, the output of the current run is compared with the *reference*. A difference between the two outputs indicates an error has resulted in a wrong output and that the error was not detected by any of the error detection mechanisms. This would contribute to the lack of coverage of the mechanism.

The following test programs were used for the experiments.

1. Test1: This test program has all the C language branch constructs (if, for, while, case).
2. Test2: SPEC benchmark (Benchmark Release 1.0) 008.*espresso*. *Espresso* is an integer benchmark and performs set operations such as union, intersect, and difference for the minimization of Boolean functions. It takes as input a Boolean function and produces a logically equivalent function possibly with fewer terms. Both the input and output functions are represented as truth tables. Approximately 50 source files comprise Espresso's body of source code.
3. Test3: SPEC benchmark (Benchmark Release 1.0) 022.*li*. This is a CPU intensive integer benchmark (basically, a LISP interpreter written in C) and performs minimal I/O. The input file to the interpreter contains Lisp code that defines the 8-queens problem. Due to the extensive time it takes for computing the solution to a single instance of the problem on a SUN 4 workstation and because of the

[1] Fault location is also called target address.

TABLE 4
Error Detection Coverage for *espresso* without ECCA

Error Models	Detection Coverage without ECCA				
	System Detection	Timeouts	User Detection	Undetected	Overall Coverage
AddIF	76.4%	1.9%	12.6%	9.1%	90.9%
AddIF2	90.1%	0.7%	5.4%	3.8%	96.2%
AddOF	91.6%	0.9%	4.7%	2.8%	97.2%
AddOS	88.8%	0.8%	7.3%	3.0%	97%
DataIF	86.9%	1%	8%	4.1%	95.9%
DataOF	77.8%	1.8%	12.2%	8.2%	91.8%
DataOS	80.4%	3.7%	10.6%	5.3%	94.7%
CndCR	26.7%	0%	53.3%	20%	80%

TABLE 5
Error Detection Coverage for *espresso* with ECCA

Error Models	Detection Coverage with ECCA					
	System Detection	Timeouts	User Detection	ECCA	Undetected	Overall Coverage
AddIF	32.8 %	0.5%	10.6%	55%	1.1%	98.9%
AddIF2	65%	0.8%	2.5%	30.8%	0.8%	99.2%
AddOF	69.4 %	0.9%	1.3%	27.8%	0.6%	99.4%
AddOS	61.3%	0.3%	3.4%	34.3%	0.6%	99.4%
DataIF	47%	0%	3.8%	48.1%	1.1%	98.9%
DataOF	35.1%	0.9%	5.1%	56.4%	2.5%	97.5%
DataOS	17.3%	1.1%	7.1%	73%	1.5%	98.5%
CndCR	25%	0%	0%	75%	0%	100%

necessity to execute it a large number of times for our fault injection experiments, the input file was modified to the 4-queens problem.

Tables 2 and 3 present the fault injection results for Test1 program with and without ECCA enabled. As can be observed from tables, the ECCA mechanism is quite powerful and increases the overall error detection coverage by about 20-25 percent. The ECCA, in some cases, has a very large error detection coverage influenced by the type of fault injected. We also observed that ECCA provides better coverage than CCA; however, that has been a foregone conclusion even before the experiments were conducted.

The results for *espresso* are given in Tables 4 and 5. The last row in the table (which is fault injection into the processor status register) is misleading, as only four faults of the entire injection campaign caused errors and the rest resulted in dormant errors. Of these four errors, three were caught by ECCA and one was a bus error. More fault injection experiments need to be performed to obtain more accurate coverage values under this fault model.

One peculiar aspect of the injection on the **022.li** benchmark is the relatively large percentage of time-out errors that occurred when injected with transient faults. As seen in Tables 6 and 7, the ECCA mechanism has also indirectly reduced the percentage of time-out errors.

As in the case of the *espresso* experiments, the fault injection results given for the *CndCR* fault model is statistically incorrect as only a limited number of the faults injected resulted in nondormant errors. One may have to run at least a million injections for that model to obtain more accurate numbers.

Note that the purpose of the second and third tests was to see how the coverage of the detection mechanisms will change when applied to larger application programs. From the results presented in the section it has been obvious that the proposed techniques provide very high coverage even for large programs such as the *SPEC* benchmarks.

4 Conclusions and Future Research

In this paper, we have presented the concepts and implementation of high and intermediate level software based approaches for on-line control flow monitoring. Fault and error injection experiments were conducted to determine the coverage provided by ECCA. The experiments showed that the ECCA technique require minimal memory and performance overhead. Error detection coverage measured for the selected programs including the *SPEC* benchmark programs were was 98 percent. The results

TABLE 6
Error Detection Coverage for 022.*li* without ECCA

Error Models	Detection Coverage without ECCA				
	System Detection	Timeouts	User Detection	Undetected	Overall Coverage
AddIF	74.3%	18.7%	1.9%	3.2%	96.8%
AddIF2	87.3%	8.6%	0.8%	3.8%	96.2%
AddOF	85.6%	10.6%	0.3%	3.5%	96.5%
AddOS	78.8%	12.8%	3.6%	4.8%	95.2%
DataIF	85.5%	11.2%	1.4%	1.9%	98.1%
DataOF	72.5%	19.5%	3.2%	4.8%	95.2%
DataOS	72.3%	18.8%	3.9%	5%	95%
CndCR	56.7%	36.7%	0%	6.7%	93.3%

TABLE 7
Error Detection Coverage for 022.*li* with ECCA

Error Models	Detection Coverage with ECCA					
	System Detection	Timeouts	User Detection	ECCA	Undetected	Overall Coverage
AddIF	50.2%	10.6%	1.5%	36.5%	1.2%	98.8%
AddIF2	74.1%	4.4%	0.6%	19.1%	1.8%	98.2%
AddOF	65.4 %	7.2%	2%	23.6%	1.8%	98.2%
AddOS	68.8%	8.2%	1.4%	18.4%	3.2%	96.8%
DataIF	61.7%	4.2%	0.9%	31.7%	1.4%	98.6%
DataOF	54.3%	14.9%	1.7%	27.2%	1.9%	98.1%
DataOS	44.3%	12.1%	3.1%	37.5%	2.9%	97.1%
CndCR	27.7%	16.7%	0%	55.6%	0%	100%

confirmed that the ECCA technique is capable of detecting data line, address line, control line, and register errors.

Albeit the low overhead required by the proposed technique, it has been observed that considerable reduction in memory and performance overhead can be achieved for ECCA through selective insertion of the assertions which makes it ideal for real-time systems. Moreover, ECCA's architectural portability allows its implementation on heterogeneous distributed systems. Research is underway to develop systematic methods to place the assertions in order to maximize error coverage and minimize overhead, with the help of analytical evaluations and experimental studies using FERRARI [1].

ACKNOWLEDGMENTS

This work was done while N. Krishnamurthy was with the Computer Engineering Research Center at the University of Texas at Austin.

REFERENCES

[1] G. Kanawati, N. Kanawati, and J. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System," *IEEE Trans. Computers,* Feb. 1995.

[2] M. Lyu, *Software Fault Tolerance.* Willy, 1995.

[3] G. Miremadi, J. Ohlsson, M. Rimen, and J. Karlsson, "Use of Time and Address Signatures for Control Flow Checking," *Fifth Int'l Working Conf. Dependable Computing for Critical Applications,* Sept. 1995.

[4] N.R. Saxena and E.J. McClusky, "Control Flow Checking Using Watchdog Assists and Extended-Precision Checksums," *IEEE Trans. Computers,* Apr. 1990.

[5] B. Ramamurthy and S.J. Upadhyaya, "Watchdog Processor-Assisted Fast Recovery in Distributed Systems," *Proc. Fifth Int'l Working Conf. Dependable Computing for Critical Applications,* Sept. 1995.

[6] M. Namjoo, "Techniques for Concurrent Testing of VLSI Processor Operation," *IEEE Trans. Computers,* 1982.

[7] A. Mahmood and E. McCluskey, "Concurrent Error Detection Using Watchdog Processors," *IEEE Trans. Computers,* Feb. 1988.

[8] K. Wilken and J. Shen, "Continuous Signature Monitoring: Low-Cost Concurrent-Detection of Processor Control Errors," *IEEE Trans. Computer-Aided Design,* June 1990.

[9] D.J. Lu, "Watchdog Processors and Structural Integrity Checking," *IEEE Trans. Computers,* July 1982.

[10] L. Mcfearin and V.S.S. Nair, "Control-Flow Checking Using Assertions," *Proc. IFIP Int'l Working Conf. Dependable Computing for Critical Applications,* Sept. 1995.

[11] K. Kanawati, N. Krishnamurthy, S. Nair, and J.A. Abraham, "Evaluation of Integrated System-Level Checks for On-Line Error Detection," *Proc. IEEE Int'l Symp. Parallel and Distributed Systems,* Sept. 1996.

[12] K.H. Huang and J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers,* June 1984.

[13] V.S.S. Nair and S. Venkatesan, "Algorithm-Based Fault Tolerance for Non-Computationally Intensive Applications," *Proc. SPIE Advanced Algorithms and Architectures for Signal Processing Conf.*, July 1994.

[14] V.S.S. Nair, H. Kim, N. Krishnamurthy, and J.A. Abraham, "Design and Evaluation of Automated High-Level Checks for Signal Processing Applications," *Proc. SPIE Advanced Algorithms and Architectures for Signal Processing Conf.*, Aug. 1996.

[15] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1988.

[16] D. Andrews, "Using Executable Assertions for Testing and Fault Tolerance," *Proc. Ninth Int'l Symp. Fault-Tolerant Computing,* June 1979.

[17] K.A. Hua, "Design of Systems with Concurrent Error Detection Using Software Redundancy," PhD thesis, Univ. of Illinois, Urbana, 1987.

[18] "Using and Porting GNU CC version 2,"Free Software Foundation, June 1991.

[19] M. Namjoo, "Concurrent Testing Using Path Signature Analysis," Stanford Univ. Technical Report CRC TR 82-16, 1982.

Zeyad AlKhalifa graduated in 1991 from George Washington University, Washington D.C., with a BS in electrical engineering while working in the Research Department of the Information Office of the Royal Embassy of Saudi Arabia. He then graduated with an MS in computer science from Shippensburg University of Pennsylvania in 1993. After holding a lecturer position for one year at the Buraydah Institute of Technology, Saudi Arabia, he returned to the U.S.A. to pursue a PhD degree in computer science at The Southern Methodist University, Dallas, Texas. His research interests include fault-tolerant computing, error correcting codes, and video transmission over ATM networks.

Suku Nair received his bachelors degree in electronics and communication engineering from the University of Kerala. He received his MS and PhD in electrical and computer engineering from the University of Illinois at Urbana-Champaign in 1988 and 1990, respectively. Currently, he is an associate professor in the Computer Science and Engineering Department at The Southern Methodist University in Dallas, where he holds a J. Lindsay Embrey Trustee Professorship in Engineering. His research interests include fault-tolerant computing and communication, VLSI systems, and software engineering. He is a member of the IEEE and ACM.

Narayanan Krishnamurthy graduated from the Indian Institute of Technology, Kharagpur, India with a BTech (Hons.) in instrumentation engineering (EE). He worked as a computer control engineer from 1988 to 1991 and as a software development engineer from 1991 to 1994. He then graduated with a Master's degree in electrical and computer engineering from The University of Texas at Austin in 1996. Currently, he is working as a CAD engineer in the Test and Logic Verification group at the Somerset PowerPC Design Center, Motorola in Austin, Texas. He is also simultaneously pursuing his PhD in electrical and computer engineering at The University of Texas at Austin. His research interests include VLSI and dependable systems design, CAD tools development/design, formal verification techniques, and software engineering/testing.

Jacob A. Abraham received the Bachelor's degree in electrical engineering from the University of Kerala, India, in 1970. His MS degree, in electrical engineering and PhD in electrical engineering and computer science, were received from Stanford University, California, in 1971 and 1974, respectively. He is a professor in the Departments of Computer Sciences and Electrical and Computer Engineering at the University of Texas at Austin. He is also director of the Computer Engineering Research Center and holds a Cockrell Family Regents Chair in Engineering. From 1975 to 1988, he was on the faculty of the University of Illinois, Urbana-Champaign.

Professor Abraham's research interests include VLSI design and test, formal verification, and fault-tolerant computing. He is the principal investigator of several contracts and grants in these areas and a consultant to industry and government on testing and fault-tolerant computing. He has published more than 200 papers and has supervised more than 40 PhD dissertations. He was elected a fellow of the IEEE in 1985 and is also a member of the ACM and Sigma Xi. He has served as an associate editor of the *IEEE Transactions on Computer-Aided Design* and the *IEEE Transactions on VLSI Systems*, and was chair of the IEEE Computer Society Technical Committee on Fault-Tolerant Computing in 1992 and 1993.