IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 13, 2018, accepted June 16, 2018, date of publication July 6, 2018, date of current version August 20, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2852805

WCET-Aware Control Flow Checking With Super-Nodes for Resource-Constrained Embedded Systems

MING ZHANG[1], ZONGHUA GU[1], (Member, IEEE), HONG LI[1], AND NENGGAN ZHENG[2], (Member, IEEE)
[1]College of Computer Science, Zhejiang University, Hangzhou 310027, China
[2]Qiushi Academy for Advanced Studies, Zhejiang University, Hangzhou 310027, China

Corresponding author: Nenggan Zheng (zng@cs.zju.edu.cn)

This work was supported by the National Science Foundation of China under Grants 61672454 and 61572433.

ABSTRACT Safety-critical embedded systems in application domains, such as aerospace, automotive, and industrial automation, must satisfy dual requirements of fault-tolerance and real-time predictability. Control flow checking is an effective technique for improving embedded systems' reliability by online monitoring and checking of software control flow to detect runtime deviations from the control flow graph. However, inserting instrumentation code in every basic block incurs significant execution time overhead, which may cause the program to violate its timing constraints. In this paper, we propose to selectively instrument a subset of code regions that are larger than basic blocks, called super-nodes, in order to make the program partially resilient to control flow errors while keeping the program worst-case execution time (WCET) below a given upper bound. WCET analysis is invoked to estimate the program WCET and to identify the corresponding worst-case execution path (WCEP). An ILP formulation is used to judiciously select a subset of super-nodes on the WCEP for instrumentation, so that the best fault detection coverage is achieved without violating the given WCET upper bound. The optimization is repeated for each identified WCEP until the program WCET satisfies the WCET upper bound. Experimental results demonstrate significant improvements of fault detection coverage compared with related work.

INDEX TERMS Control flow checking, real-time embedded systems, soft errors.

I. INTRODUCTION

Real-time embedded systems in many application domains, such as aerospace, automotive and industrial automation are often safety-critical, and their malfunction may lead to catastrophic consequences, including loss of human life. As semiconductor technology scales down further, decreasing size, capacitance and threshold voltage level of devices allow better performance and lower power consumption, but also make devices more susceptible to transient faults, especially for embedded systems operating in harsh environments, such as aerospace or automotive systems. Specifically, we focus on transient faults caused by harsh environmental conditions such as intensive space radiation, and assume the common Single-Event Upset (SEU) fault model.

For protecting against soft errors in the memory subsystem, including main memory and cache, Error-Correcting Code (ECC) is an effective mechanism. In fact, ECC is a mandatory requirement for memory components used in safety-critical application domains, such as satellite and aerospace applications. But other mechanisms are needed to harden the processor pipeline against transient faults. One approach is to use Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR), which executes redundant copies of the same program on two or three identical processing elements and compares their outputs, to detect both control flow and dataflow errors. Another approach is to use radiation-hardened processors that are resistant to space radiations. Such approaches are very effective in achieving fault-tolerance, and are suitable for safety-critical applications that can afford the cost, but may not be feasible for mass-produced consumer products such as automotive electronics that are highly cost-sensitive, and the designer must resort to software and compiler techniques to achieve fault tolerance. Another motivating application scenario is

pico-satellites based on Commercial Off-The-Shelf (COTS) smartphones [1], which must achieve fault-tolerance with low cost commercial hardware.

Transient faults may cause either Control Flow Errors (CFEs) or dataflow errors. A CFE disrupts normal control flow of a program by causing the program counter to jump to an erroneous location, whereas a dataflow error affects numerical results of computation without altering the control flow. Compared with dataflow errors, CFEs are more likely to cause severe failures such as program crash and hang, and are the focus of this paper.

A software program consists of a number of Basic Blocks (BBs). A BB is a straight line of instructions that are always executed from the beginning to the end at runtime, without any Control Flow Instructions (CFIs) in the middle. A program's Control Flow Graph (CFG) has nodes representing BBs and edges representing valid control flow between the BBs. A CFE occurs when runtime control flow deviates from the program CFG. There may be three types of CFEs: *branch insertion error*: a non-control flow instruction is changed to a CFI; *branch deletion error*: a control flow instruction is changed to a non-control flow instruction; *branch target modification*: target address of a control flow instruction is modified. Many *Control Flow Checking (CFC)* techniques [2], [3] have been proposed to detect and/or correct CFEs caused by transient hardware faults, both hardware-based and software-based. Hardware-based CFC techniques incur low runtime overhead but require modification to commodity hardware. Software-based CFC does not require hardware modification, but the additional instrumentation code typically incurs large performance overhead.

For real-time systems, correctness depends not only on producing correct results, but also on correct timing of delivering the results. To make system-level timing guarantees, it is a prerequisite to provide an upper bound on each individual task's Worst-Case Execution Time (WCET) [1], which is much more important than its Average-Case Execution Time (ACET). Safety-critical embedded systems must satisfy the dual requirements of timing predictability and fault-tolerance simultaneously. Previous work has focused on optimizing the average-case performance, but has not adequately addressed WCET requirements in hard real-time systems. In order to achieve practical deployment of software-based CFC techniques on resource-constrained hard real-time systems, Gu *et al.* [5] presented *WCET-Aware Control Flow Checking (WACFC)*, which allows flexible trade-offs between fault detection coverage and program WCET by instrumenting a select subset of BBs in the CFG. In this paper, we present *WCET-Aware Control Flow Checking with Super-Nodes (WACFC-SN)* by extending WACFC in [5] to use the concept of *super-nodes* to further improve efficiency and fault coverage.

Table 1 summarizes the main abbreviations used in this paper.

This paper is organized as follows: Section II discusses related work, including WCET analysis and optimization, software-based CFC, super-node-based CFC and WCET-aware partial CFC; Section III presents the details of our proposed algorithm WACFC-SN; Section IV presents experimental results, and Section V concludes the paper.

TABLE 1. Abbreviations.

ACET	Average-Case Execution Time
BB	Basic Block
CFC	Control Flow Checking
CFE	Control Flow Error
CFG	Control Flow Graph
CFI	Control Flow Instruction
OH_{inst}	CFC Instrumentation Overhead in each basic block or super-node
SEU	Single-Event Upset
WACFC	WCET-Aware Control Flow Checking
WACFC-BB	WACFC based on Basic Blocks
WACFC-SN	WACFC based on Super-Nodes
WCEP	Worst-Case Execution Path
WCET	Worst-Case Execution Time
$WCET_{orig}$	WCET of the original program without CFC
$WCET_{FCFC}$	WCET of the program with full CFC
$WCET_{PCFC}$	WCET of the program with partial CFC
$WCET_{ub}$	WCET upper bound specified by the designer

II. BACKGROUND AND RELATED WORK

A. SYSTEM-LEVEL FAULT-TOLERANCE TECHNIQUES

Researchers have addressed the intersection between real-time systems and fault-tolerance at the system-level, where multiple tasks are managed by a real-time operating system. Zhu [6] presented reliability-aware energy management schemes that dynamically schedule error recoveries for tasks to compensate for reliability loss due to dynamic voltage and frequency scaling. They also exploited checkpoint techniques to efficiently use the runtime timing slack. Li *et al.* [8] presented a reliability-guaranteed energy-aware frame-based task set execution strategy for hard real-time systems, where processing resources are reserved to re-execute tasks when transient faults occur. Li *et al.* [7] presented algorithms for runtime reconfiguration of a defect-tolerant many-core Network-on-Chip when defective cores are replaced by backup cores. The above papers generally assume each task's WCET is given as problem input, and address system-level multi-tasking issues.

B. WCET ANALYSIS AND OPTIMIZATION

WCET analysis consists of two steps [4]: the low-level analysis step determines execution time of BBs, taking into account effects of micro-architectural structures (e.g., cache, pipeline, etc.); the high-level analysis step takes the execution time of BBs as input and determines the global WCET of the program according to its CFG. Since exhaustive enumeration of all program execution paths is computationally infeasible, the prevalent approach to WCET analysis is the *Implicit Path Enumeration Technique (IPET)* [9] based on an ILP formulation.

A program's WCET is determined by the Worst-Case Execution Path (WCEP), defined as the program execution

Algorithm 1 Global Super-Node Selection

$N = \emptyset$, $Overlap\,(b) = \emptyset$, $N_{selected} = \emptyset$, $N_{deadlock} = \emptyset$
for each function f
    Find all super-nodes in f and add them to N
    for each BB b in f
        Add all super nodes in f that contains b to $Overlap\,(b)$
    end for
end for
Mark every singleton super-node as instrumented
$B_{covered} = B$, $N_{candidate} = N$
Find the current $WCEP = \{(b, ec_b)\}$ and the current $WCET$
while $WCET > WCET_{ub}$
    Select a set S of super-nodes from $N_{candidate} \cap N_{WCEP}$ by invoking the Path-local Super-node Selection algorithm
    Mark each super-node $n \in S$ as instrumented and other super-nodes on the current path as un-instrumented
    $B_{covered} = (B_{covered} \backslash B_{WCEP}) \cup \{h_n | n \in S\}$
    Mark as un-instrumented and remove from $N_{candidate}$ super-nodes that conflict with any $n \in S$
    if no super-node is removed from $N_{candidate}$
        for each $n \in N_{WCEP}$
            if $n \in N_{deadlock}$ **then**
                if the selection state of n changes **then**
                    $sc_n = sc_n + 1$ **end if**
            else $sc_n = 0$ **end if**
        end for
        if $N_{WCEP} \cap N_{candidate} \not\subseteq N_{deadlock}$ **then**
            $N_{deadlock} = N_{deadlock} \cup (N_{WCEP} \cap N_{candidate})$
        else
            Mark as un-instrumented and remove from $N_{candidate}$ the smallest super-node n with the maximum value of sc_n.
            $B_{covered} = B_{covered} \backslash h_n$
            $N_{deadlock} = \emptyset$
        end if
    else $N_{deadlock} = \emptyset$ **end if**
    Find the current $WCEP = \{(b, ec_b)\}$ and the current $WCET$
end while
$N_{selected} = \{innermost\ n \in N_{candidate}\ s.t.\ \exists b \in B_{covered}: b \in B_n\}$
Actually instrument every $n \in N_{selected}$

path with the largest possible execution time. In general, the WCEP may be a rarely-executed "cold path" that has a very small probability of occurring at runtime, but it is still important to account for it when computing the WCET for hard real-time systems. By contrast, other shorter paths with smaller execution times may be frequently-executed "hot paths" that determine the Average-Case Execution Time (ACET), but do not contribute to the WCET. Whereas conventional compiler optimization algorithms aim to reduce ACET, Suhendra *et al.* [10] and Wang *et al.* [11],[12] presented compiler optimization algorithms that aim to decrease program WCET by allocating program data variables to on-chip scratchpad memory. A challenge in WCET reduction is the so-called *WCEP switch*, i.e. after reducing the execution time of the current WCEP, a new path may become the new WCEP that determines the WCET. Therefore, the designer needs to optimize each identified WCEP as the current WCEP switches among different paths during the optimization process, until the program WCET falls below $WCET_{ub}$. References [10]–[12] start with a program with all data variables in off-chip main memory, and gradually allocate data variables in each BB to on-chip scratchpad memory to reduce program WCET. Similarly, we start with a program with full Control Flow Checking, and gradually reduce its WCET by instrumenting only a selected subset of super-nodes, until the desired WCET upper bound is reached.

C. SOFTWARE-BASED CONTROL FLOW CHECKING

Software-based CFC techniques [2] typically work by inserting additional checking code, called *instrumentation code*, at the beginning and end of each BB. Each BB has associated entry and exit signatures computed offline to encode some invariants derived from the CFG. As control flow passes through a BB at runtime, the instrumentation code in each BB computes the signatures, typically with bitwise AND/XOR operations, and compares the runtime signatures with the pre-computed expected signatures. In case of a mismatch, a CFE is detected and the error handler is invoked. Different CFC algorithms differ in the technical details of computing and checking signatures, but the overall framework is the same. Our approach in this paper is independent of the specific details of the CFC algorithm adopted, and is broadly applicable to other CFC algorithms.

Although only a few instructions are added to each BB, the total runtime overhead of CFC is quite significant, since most BBs contain only a few instructions (e.g., 3-6 for typical MIPS programs) on average for typical non-scientific program workloads [13]. The high performance overhead of software-based CFC makes it unsuitable for resource-constrained embedded systems. Vemu and Abraham [14] and Khudia and Mahlke [15] have presented techniques for reducing runtime overhead of software-based CFC, but they focused on average-case performance instead of worst-case performance as measured by the program WCET, which is more important to hard real-time systems.

D. SUPER-NODE-BASED CFC

In order to reduce overhead, Vemu and Abraham [14] proposed *node expansion*, i.e., to insert instrumentation code only to the beginning and end of each *super-node* instead

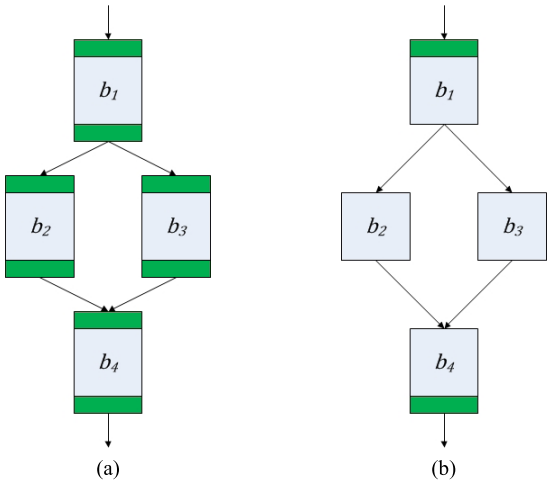


FIGURE 1. Basic block instrumentation vs. super-node instrumentation. Solid shade denotes instrumentation code.

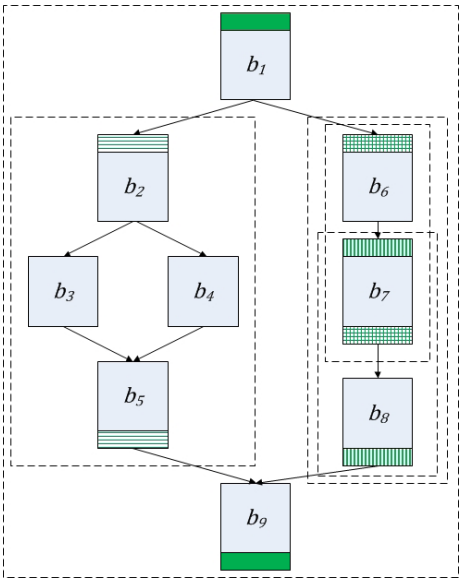


FIGURE 2. Relationships between super-nodes and their impacts on instrumentation.

of each BB (Fig. 1 (b)), thus reducing the amount of instrumentation code and its runtime overhead. A super-node is a single-entry single-exit code region, where each execution of the header must be followed by one execution of the footer, and vice versa. A super-node is said to be on an execution path if its header (and footer) is on the path. With super-nodes based CFC, CFEs within each super-node cannot be detected; only CFEs between different super-nodes can be detected.

Here are the basic definitions of super-nodes:

1) DOMINATOR

A BB b_1 is a dominator of a BB b_2 if every path from the start of the program that reaches b_2 has to pass through b_1.

2) POSTDOMINATOR

A BB b_1 is a postdominator of a BB b_2 if every path from b_2 to the end of the program has to pass through b_1.

3) DOMINATOR-POSTDOMINATOR PAIR

BBs b_1 and b_2 form an ordered dominator-postdominator pair (b_1, b_2) iff b_1 is a dominator of b_2 and b_2 is a postdominator of b_1. If b_1 and b_2 form a dominator-postdominator pair, b_1 and b_2 are different BBs, and b_1 is in every loop that b_2 is part of and vice versa, then b_1 and b_2 form a **strict dominator-postdominator pair**.

4) SUPER-NODE

A super-node is formed from a strict dominator-postdominator pair (b_1, b_2), where BBs b_1 and b_2 are the *header* and *footer* of the super-node, respectively; every BB on every path from b_1 to b_2 (including b_1 and b_2) is part of the super-node.

We use (b_i, b_j) to denote the super-node corresponding to the dominator-postdominator pair(b_i, b_j). A super-node *contains* or *covers* a BB if the BB is part of the super-node. As a special case, each BB is also a *singleton super-node*.

Super-nodes are different from *super blocks* in compiler optimization [17]. A super-block is an execution trace, i.e., a linear sequence of BBs with no side entrances. Control flow may only enter from the top but may leave at one or more exit points. In contrast, a super-node consists of a directed graph of BBs with possible control flow within the super-node. If a BB in a super-node is on an execution path, only the header and footer of the super-node are guaranteed to be on the execution path, while the other BBs may or may not be on the path depending on runtime control flow decisions.

Two different super-nodes n_i and n_j may be either *disjoint* or *overlapping* (Fig. 2). If n_i and n_j contain at least one common BB, then n_i and n_j *overlap* with each other (e.g., (b_1, b_9) and (b_2, b_5); (b_6, b_7) and (b_7, b_8)); otherwise, they are *disjoint* (e.g., (b_2, b_5) and (b_6, b_7)). For two overlapping super-nodes n_i and n_j, if n_j is a true subset of n_i, then n_i *includes* n_j (e.g., (b_1, b_9) includes (b_2, b_5)). We refer to n_i as an *outer node* of n_j, and n_j as an *inner node* of n_i. If neither of two different overlapping super-nodes includes the other, then the super-nodes *partially overlap* each other (e.g., (b_6, b_7) and (b_7, b_8)). A super-node n_j is said to *conflict with* another super-node n_i if n_i includes or partially overlaps n_j.

E. WCET-AWARE PARTIAL CFC

Assuming that WCET of the original program without CFC instrumentation, $\text{WCET}_{\text{orig}}$, does not exceed WCET_{ub}; and $\text{WCET}_{\text{FCFC}}$, WCET of the program with full CFC instrumentation on all BBs, exceeds WCET_{ub}, Gu *et al.* [5] presented WCET-aware partial CFC that allows flexible tradeoffs between fault detection coverage and program WCET, and provides a tunable fault detection technique that can be adapted by the designer to suit the needs of different applications. It works by selectively instrumenting a subset of BBs in the program to endow the program with partial fault detection capability while keeping its WCET within a given

upper bound WCET_{ub}. The objective is to achieve *reasonable* fault detection coverage within a *reasonable* upper bound on the program WCET, where the notion of *reasonable* should be application-dependent and specified by the designer. If the processor workload is light, or the processor has enough computing power, then more BBs can be instrumented to achieve higher fault detection coverage; if the system is overloaded, then the designer may choose to instrument fewer BBs to reduce the workload. However, partial CFC may not be applicable to certain highly safety-critical applications, e.g., avionics control systems, and full CFC must be used even if it is necessary to add more processors or switch to a faster processor to reduce the per-processor workload.

III. WCET-AWARE CONTROL FLOW CHECKING WITH SUPER-NODES

We now present our main contribution, the WACFC-SN algorithm, which consists of two parts: *Global Super-node Selection*, a heuristic algorithm, and *Path-local Super-node Selection*, an ILP formulation. We start with a program with full CFC, and WCET exceeding the designer-specified upper bound WCET_{ub}. During each iteration, we perform WCET analysis to obtain the program WCET and identify the current WCEP. If the current WCET exceeds WCET_{ub}, we invoke the Path-local Super-node Selection algorithm to reduce execution time of the WCEP by instrumenting only a selected subset of super-nodes on the current path. After this step, a new path may become the WCEP, so we perform WCET analysis again, and repeat this process until the program WCET falls below WCET_{ub}. We first present the definition of CFE Detection Capability, then the Path-local Super-node Selection algorithm, and finally the Global Super-node Selection algorithm.

A. CFE DETECTION CAPABILITY

The probability of transforming a non-branch opcode to a branch opcode (and vice versa) due to a single bit flip is generally assumed to be very low [15], [16]. We adopt the same assumption in this paper, and focus on CFEs due to branch target modification.

When a CFE occurs, control flow is directed from a source CFI to an erroneous target instruction. Transient faults may occur anywhere in the processor pipeline, from instruction fetch, to decode, to execute, and to retirement. Consider a CFI, e.g., a branch or jump instruction. The instruction may have a field encoding the target address. As long as the instruction stays in the processor pipeline, it may be affected by a transient fault, such as a bit-flip in the pipeline register where the target address is stored, which may cause a CFE. Therefore, the longer a CFI stays in the processor pipeline, the more vulnerable to soft errors it becomes. The total time a CFI spends in the processor pipeline during a program run depends on both the instruction execution time (i.e., the time each execution of the instruction takes) and the execution frequency of the instruction. For any given CFI, its execution frequency determines its total execution time. We assume that the probability for a CFI i to experience a CFE at runtime is proportional to the average-case execution frequency F_i of the CFI. By definition, each BB b contains at most a single CFI, hence the average-case execution frequency of a CFI is equal to the average-case execution frequency F_b of its BB that contains it. We collect the average-case execution frequency F_b of every BB by profiling the program with random inputs.

We now consider the destination of the CFE, i.e., the code region that is the target of invalid control flow due to the CFE. A large super-node that spans a large range of memory addresses is more likely to become the destination of a CFE than a smaller super-node, assuming that all memory addresses have equal probability of being the target of a CFE. For a RISC processor where all instructions have the same length (number of bits), we define the size of a given super-node n as the total number of instructions in all BBs B_n contained in it:

$$\sum_{b \in B_n} IC_b \tag{1}$$

where IC_b is the instruction count, i.e., number of instructions in BB b. We assume the probability for a super-node n to become the destination of a CFE is proportional to the size of the super-node.

For a given super-node n, the probability for a CFE that jumps from a CFI outside the super-node to some instruction inside the super-node is proportional to the average-case execution frequency of the source CFI times the size of the destination super-node:

$$F_i \cdot \sum_{b \in B_n} IC_b \tag{2}$$

CFC instrumentation of a super-node enables detection of CFEs that direct control flow from a CFI outside the instrumented super-node to some instruction inside that super-node, but not vice versa. In each instrumented super-node, instrumentation code is added to both the beginning of the header and the end of the footer to update the runtime signature and check it against a pre-computed expected signature. At compile-time, a unique expected signature is assigned to the inside of each instrumented super-node. If no CFE occurs, the runtime signature is updated to the expected signature at the beginning of the header of the super-node, which is expected by the signature check at the end of the footer. If a CFE directs control flow from a CFI outside the super-node to the inside of the super-node, the runtime signature is not updated at the beginning of the header of the super-node, and thus will not match the expected signature for the inside of the super-node. Such a mismatch will be caught by the signature check at the end of the footer of the super-node and the CFE is detected. If a CFE directs control flow to an instruction outside any super-node, control flow may or may not pass through a signature check after the CFE occurs. Furthermore, if a CFE directs control flow between two instructions that are not inside any super-node, the expected signatures at the

instructions may be the same. In either case, the CFE may go undetected.

The observation above motivates our definition of *Super-node CFE Detection Capability* D_n of super-node n as a metric to estimate the fault detection coverage provided by an instrumented super-node n:

$$D_n = \left(\sum_{b \in B - B_n} F_b \right) \cdot \sum_{b \in B_n} IC_b \tag{3}$$

where B is the set of all BBs in the program code, B_n is the set of BBs in super-node n. Intuitively, super-node n's CFE detection capability is proportional to its size, i.e., number of instructions *inside* the super-node, and the sum of average-case execution frequencies of all CFIs *outside* the super-node.

To estimate the fault detection coverage of an execution path p with a selected set $N^p_{selected}$ of instrumented super-nodes, the *Path CFE Detection Capability* D_p of the path is defined as the sum of Super-node CFE Detection Capabilities of all instrumented super-nodes on the path:

$$D_p = \sum_{n \in N^p_{selected}} D_n \tag{4}$$

Similarly, for a program with a selected set $N_{selected}$ of instrumented super-nodes, the *Program CFE Detection Capability* D can be defined as the sum of Super-node CFE Detection Capabilities of all instrumented super-nodes in the program code:

$$D = \sum_{n \in N_{selected}} D_n \tag{5}$$

This Program CFE Detection Capability serves as a proxy for the actual fault detection coverage, which can only be obtained through fault injection experiments, but not expressible analytically. Experimental results confirm that this objective is more effective than the objective of maximizing the total average execution time of instrumented BBs in the program in [5].

B. ILP FORMULATION FOR PATH-LOCAL SUPER-NODE SELECTION

In this section, we present the ILP formulation for the Path-local Super-node Selection algorithm, which selects a subset of candidate super-nodes $N_{candidate}$ on the current WCEP for instrumentation to maximize the Path CFE Detection Capability while keeping the execution time of the path below WCET_{ub}. For each candidate super-node n on the current WCEP, a 0-1 variable x_n is associated with it to indicate whether the super-node is selected for instrumentation ($x_n = 1$) or not ($x_n = 0$). The optimization objective is to maximize the Path CFE Detection Capability of the current WCEP:

$$\sum_{n \in N_{candidate} \cap N_{WCEP}} D_n \cdot x_n \tag{6}$$

where N_{WCEP} denotes the set of super-nodes on the current WCEP. This definition of the Path CFE Detection Capability is equivalent to (4).

One constraint is that execution time of the current WCEP must not exceed WCET_{ub}:

$$WCET_{orig} + \sum_{n \in N_{candidate} \cap N_{WCEP}} ec_{h_n} \cdot OH_{inst} \cdot x_n \leq WCET_{ub} \tag{7}$$

where $\text{WCET}_{\text{orig}}$ denotes the original WCET of the program without any instrumentation; ec_{h_n} denotes the execution count of the header BB h_n of super-node n on the current WCEP, which is also the execution count of super-node n on the current WCEP; OH_{inst} denotes the instrumentation overhead, i.e., the additional execution time of any single super-node due to additional instrumentation code.

Another constraint is that overlapping super-nodes, i.e., super-nodes that contain the same BB, must not be selected for instrumentation simultaneously:

$$\forall b \in B: \sum_{n \in Overlap(b) \cap N_{candidate} \cap N_{WCEP}} x_n \leq 1 \tag{8}$$

where B denotes the set of all BBs in the program, and $Overlap\ (b)$ denotes the set of super-nodes that contain BB b. To see why overlapping super-nodes cannot be instrumented simultaneously, consider Fig. 2 as an example. Suppose (b_1, b_9) is instrumented. Instrumentation code in b_9 (marked with solid shade in b_9) expects a runtime signature value produced by instrumentation code in b_1 (marked with solid shade in b_1). If the super-node (b_2, b_5) is instrumented simultaneously with (b_1, b_9), instrumentation code for (b_2, b_5) (marked with horizontal lines) may execute between b_1 and b_9 and alter the runtime signature value produced by instrumentation code in b_1, causing the exit check at the end of b_9 to be invalid. The same problem happens for partially overlapping super-nodes (b_6, b_7) and (b_7, b_8).

The optimization objective (6), and constraints (7)(8) form the ILP model of the Path-local Super-node Selection algorithm. The algorithm may reduce execution time of a path in two ways. First, if more than one super-node on the path has previously been selected for instrumentation and some of the selected super-nodes are inner nodes of the same super-node, then the common outer node may be selected for instrumentation instead of its inner nodes. In this case, the program loses the ability of detecting invalid control flow between different parts within the outer node. Second, a super-node on the path which has previously been selected for instrumentation may be de-selected. In this case, the program loses the ability of detecting invalid control flow from the outside of the super-node to its inside. In both cases, the CFE Detection Capability is reduced for a decrease in execution time of the path.

It is not possible to formulate a unified ILP model to optimize (5), because WCET analysis with IPET [9] involves an ILP formulation with the objective of maximizing program execution time. Hence we cannot express the WCET upper bound constraint in (7) with a set of linear constraints in the overall ILP model with the objective of maximizing (5).

Next, we introduce the Global Super-node Selection algorithm (Algorithm 1).

C. GLOBAL SUPER-NODE SELECTION ALGORITHM

We assume a program without any CFC instrumentation satisfies the given WCET_{ub}. We first identify the set of all super-nodes (N), and the set of overlapping super-nodes $Overlap\,(b)$ which contains the same BB b for each BB (Lines 2-7). The algorithm starts with a fully instrumented program, where every BB, i.e., singleton super-node, is marked as instrumented and the instrumentation overhead OH_{inst} is added to the execution time of every singleton super-node (Line 8). The set $N_{candidate}$ of candidate super-nodes is initialized to contain all (possibly overlapping) super-nodes (Line 9). The candidate super-nodes in $N_{candidate}$ are the only super-nodes that may be selected for instrumentation by the Path-local Super-node Selection algorithm. We do not actually instrument or un-instrument super-nodes until the end of the algorithm, when the program's WCET satisfies WCET_{ub} and all super-nodes to be instrumented are determined.

If the current program WCET exceeds WCET_{ub}, then we gradually reduce the amount of instrumentation code until the program WCET falls below WCET_{ub} (Lines 10-32). We first perform WCET analysis to obtain the program WCET and the corresponding WCEP (Line 10), which consists of a set of tuples $\{(b, ec_b)\}$ denoting each BB b on the WCEP and its execution count ec_b. If the current WCET exceeds WCET_{ub} (Line 11), we select a set S of super-nodes on the current WCEP from $N_{candidate}$ by invoking the Path-local Super-node Selection algorithm (Line 12). The super-nodes selected by the Path-local Super-node Selection algorithm are marked as instrumented and other super-nodes on the current path are marked as un-instrumented (Line 13). Then, super-nodes that conflict with any of the selected super-nodes are marked as un-instrumented and removed from $N_{candidate}$ (Line 15). This ensures that the execution time of a selected super-node will not be increased by preventing its inner nodes from being selected for instrumentation. Super-nodes that do not conflict with any selected super-node remain in $N_{candidate}$ and may be selected later when a different WCEP is optimized or when the same WCEP is optimized again.

When the execution time of the current path is decreased below WCET_{ub}, we perform WCET analysis again to identify the new WCEP (Line 31). This process is repeated until the program WCET falls below WCET_{ub}, when we have found a valid solution, and can proceed to actually instrument the selected super-nodes. To keep track of the selected super-nodes, we maintain a set $B_{covered}$ of BBs, which contains headers of super-nodes that are selected for instrumentation. Initially, the set $B_{covered}$ contains all BBs of the program to reflect the fact that every singleton super-node is initially selected for instrumentation (Line 9). After every invocation of the Path-local Super-node Selection algorithm, $B_{covered}$ is updated to contain headers of the selected super-nodes on the current path as well as headers of selected super-nodes on other paths (Line 14). When the program WCET is decreased below WCET_{ub}, for each $b \in B_{covered}$, the innermost super-node $n \in N_{candidate}$ that covers b is actually instrumented (Lines 33-34).

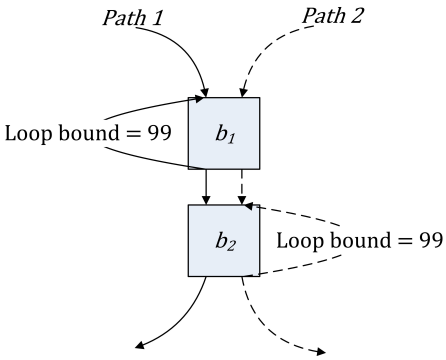


FIGURE 3. Two paths of the same program with two BBs BB1 and BB2 in common.

D. DEADLOCK PREVENTION

From Algorithm 1, we can see that the state of $N_{candidate}$ is closely related to whether the algorithm terminates or not. Every time the Global Super-node Selection algorithm optimizes a path, some candidate super-nodes may be removed from $N_{candidate}$ so that they will not be instrumented. Super-nodes are never added to $N_{candidate}$ after it is initialized. Therefore, as long as the algorithm keeps removing super-nodes from $N_{candidate}$, the set will eventually become empty, and the algorithm will terminate. However, the algorithm might stop removing super-nodes from $N_{candidate}$ and stop making progress under certain circumstances, if no mechanism were incorporated to prevent deadlocks.

As an example, consider Fig. 3. Suppose BB b_1 has execution count of 99 on *Path 1* and execution count of 1 on *Path 2*; BB b_2 has execution count of 1 on *Path 1* and execution count of 99 on *Path 2*. Starting from the configuration where both b_1 and b_2 are instrumented, we run the WCET analysis tool and find out WCET_{ub} is exceeded. Suppose *Path 1* is identified as the WCEP, and we decide to un-instrument $b_1$to reduce the execution time of *Path 1*. We re-run the WCET analysis tool and find out the WCET_{ub} is still exceeded, but the WCEP has switched to *Path 2*. In order to reduce the execution time of *Path 2*, we now decide to un-instrument b_2 and re-instrument b_1. But this causes the WCEP to switch back to *Path 1*, when the algorithm enters an infinite loop, or a deadlock.

We note three necessary conditions for causing such a deadlock.

1) A set of execution paths that pass through at least two common super-nodes are optimized repeatedly in alternation.
2) No super-node is removed from $N_{candidate}$, since super-nodes that conflict with any of the selected super-nodes have already been removed.

3) For at least two of the common super-nodes, optimization of the multiple execution paths disagree on whether the super-nodes should be selected for instrumentation or not.

To detect and prevent deadlocks, we incorporate a *Deadlock Prevention* mechanism in the Global Super-node Selection algorithm, which runs every time the algorithm finishes optimizing a path (Lines 16-30). It detects potential deadlocks by checking conditions 1) and 2) above. If no candidate super-node is removed from $N_{candidate}$ (Line 16), a deadlock may occur in the future, and the Deadlock Prevention mechanism is triggered. We maintain a set $N_{deadlock}$ containing candidate super-nodes on the paths that are recently optimized since the last time $N_{candidate}$ starts to remain unchanged. If $N_{deadlock}$ has not converged to a fixed set, i.e., a candidate super-node on the current path has not been added to $N_{deadlock}$ (Line 23), then the current path has not been optimized since the last time $N_{candidate}$ remains unchanged, and there is no deadlock. In this case, all candidate super-nodes on the current path are added to $N_{deadlock}$ (Line 24). If $N_{deadlock}$ has converged to a fixed set, then a potential deadlock is detected.

To resolve the deadlock, we can force progress by removing a super-node from $N_{candidate}$. We choose to remove the candidate super-node that is the most likely to cause disagreement. When optimization for a path disagrees with optimization for another path on whether a common super-node on the paths should be selected for instrumentation or not, it changes the selection state of the super-node, i.e., it either selects the super-node that was not previously selected for instrumentation, or de-selects the super-node that was previously selected. We assume a super-node that has experienced the most selection state changes is the most likely to cause disagreement, and maintain a state change counter sc_n for every super-node $n \in N_{deadlock}$. When $N_{candidate}$ remains unchanged and a candidate super-node n on the current path is added to $N_{deadlock}$, if the super-node has not been previously added to $N_{deadlock}$, then its state change counter sc_n is set to zero to indicate that it has not experienced any change in its selection state (Line 21). For a super-node that is already in $N_{deadlock}$, if its selection state changes, its state change counter is incremented (Lines 18-20). When a potential deadlock is detected, the super-node in $N_{deadlock}$ that has experienced the most selection state changes is marked as uninstrumented and removed from $N_{candidate}$ (Line 26) to force progress, and its header is removed from $B_{covered}$ (Line 27). Meanwhile, since the deadlock is broken, $N_{deadlock}$ is cleared (Line 28).

With the Deadlock Prevention mechanism, the Global Super-node Selection algorithm always terminates. Every time a path is optimized, the algorithm either makes progress by removing a super-node from $N_{candidate}$ (Line 15), or checks $N_{deadlock}$ to detect a potential deadlock (Line 23). If $N_{deadlock}$ converges to a fixed set, then the algorithm forces progress by removing a super-node from $N_{candidate}$ (Line 26). Otherwise, it adds all candidate super-nodes on the current path to $N_{deadlock}$ (Line 24). $N_{deadlock}$ definitely converges to a fixed set after a finite number of paths are optimized, since $N_{candidate}$ is a finite set. In the worst-case, $N_{deadlock}$ converges to $N_{candidate}$ when all super-nodes in $N_{candidate}$ are added to $N_{deadlock}$. The convergence of $N_{deadlock}$ to a fixed set will in turn force progress by removing a super-node from $N_{candidate}$. In all cases, a super-node is removed from $N_{candidate}$ after a finite number of paths are optimized, and the algorithm can always make progress.

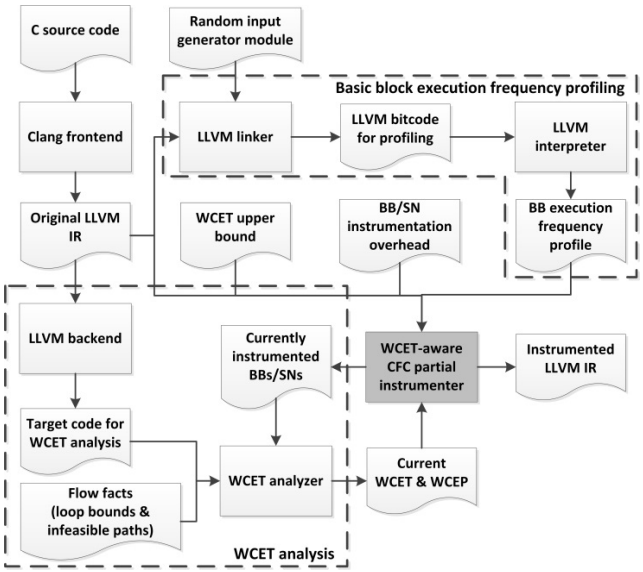


FIGURE 4. Framework of WCET-aware Partial CFC Instrumentation. The CFC partial instrumenter (shaded box) implements the proposed WACFC-SN algorithm.

IV. EVALUATION AND RESULTS

A. EXPERIMENTAL SETUP

We evaluate the WACFC-SN algorithm with fault injection experiments. Fig. 4 shows the overall experimental setup. We use the commercial WCET analyzer aiT from the company AbsInt (https://www.absint.com/). aiT takes as input a binary executable file and the hardware processor specification, and derives the program WCET estimate and the corresponding WCEP. We use the Freescale PowerPC 5554 processor, a widely-used ECU in automotive systems, and one of the processor models supported by aiT. The instrumentation overhead OH_{inst} is assumed to be 80 cycles, which is the maximum observed execution time incurred by the instrumentation code plus a safety margin.

We use a diverse collection of programs from the Mälardalen WCET benchmark suite [18] in our experiments. We use the LLVM compiler framework [19] to compile each benchmark program into LLVM Immediate Representation (IR), which is the input for further analysis and transformation. To obtain average execution frequencies of BBs, the LLVM IR is linked with a program-specific module that generates random inputs for the program at runtime. The program is run 1000 times with different random inputs, and the

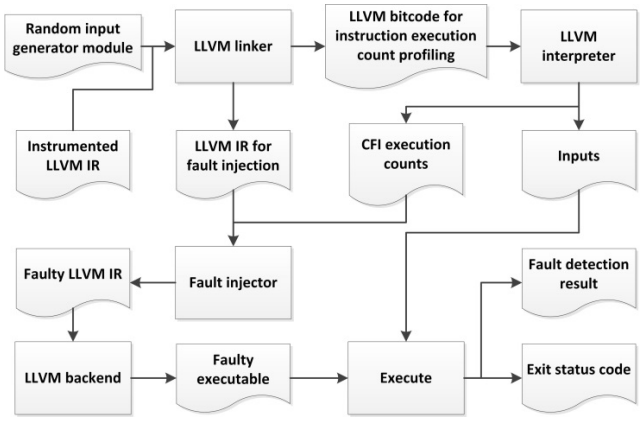


FIGURE 5. Fault-injection workflow.

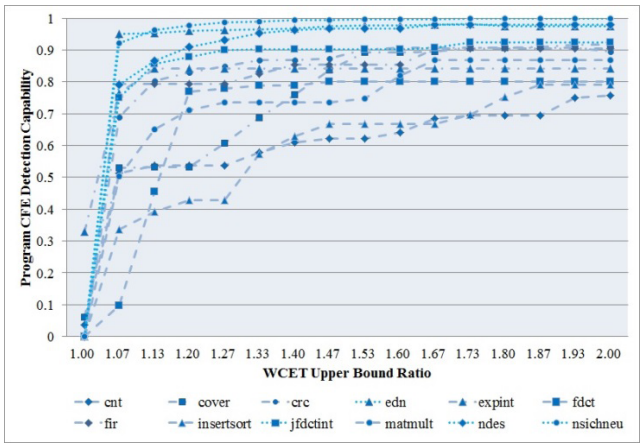


FIGURE 6. CFE Detection Capability for the benchmark programs vs. WCET upper bound ratio.

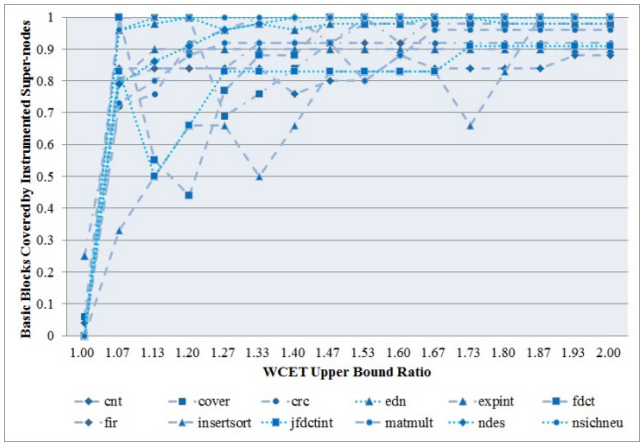


FIGURE 7. Proportion of BBs covered by instrumented super-nodes for the benchmark programs vs. WCET upper bound ratio.

execution frequency of every BB is collected using the profiler *llvm-prof*. We use the collected execution frequency of every BB as the average-case execution frequency of the BB. We then generate target binary executable from the LLVM IR by using a suitable LLVM backend in preparation for WCET analysis. The LLVM IR and the BB execution frequency profile are fed into the WCET-aware CFC partial instrumenter, along with the designer-specified $WCET_{ub}$. To facilitate comparison among different programs, we use the normalized *WCET upper bound ratio* ($WCET_{ub}/WCET_{orig}$) in the experimental results. The partial instrumenter invokes the external WCET analyzer with information about currently instrumented super-nodes to update itself with the current WCEP and WCET, and finally produces the enhanced LLVM IR with instrumentation.

We conduct fault injection experiments with an in-house fault injector, which simulates CFEs by processing the instrumented LLVM IR. The WCET-aware CFC partial instrumenter is run on each benchmark program with a series of different WCET upper bound ratios. For each one, 2000 fault injection experiments are conducted to evaluate the fault detection coverage. Fig. 5 shows workflow of each fault injection experiment. For each experiment, a single random CFE is injected into the instrumented LLVM IR to simulate the effect of a transient fault, consistent with the common SEU fault model. To ensure that each injected fault is activated, the program is run twice with the same random inputs for each fault injection experiment. During the first run, no fault is injected and the execution trace is extracted. In particular the number of executions of each BB (which is also the number of executions of the CFI in the BB) is collected in a similar way as BB execution frequency profiling. During fault injection, the fault injector chooses a random CFI, which is executed at least once on the execution trace of the first run, as the source CFI of the CFE, and ensures the CFE is activated only once during a program run. For the second run, the program with the CFE is run. The program output and the exit code are collected and examined to determine whether the CFE is detected or not.

B. EXPERIMENTAL RESULTS

For each benchmark program, we vary the WCET upper bound ratio from 1.0 to 2.0. Fig. 6 shows how the Program CFE Detection Capability varies with the WCET upper bound ratio. We can see that a significant increase in the Program CFE Detection Capability is achieved at the expense of a small increase in the program WCET for most programs. Similarly, as can be seen in Fig. 7, a significant increase in the number of BBs that are covered by instrumented super-nodes is achieved at the expense of a small increase in the program WCET for most programs. Note that a larger number of covered BBs does not imply a higher CFE Detection Capability. For example, when a large super-node is instrumented, all basic blocks in the super-node are covered. But invalid control flow between different parts within the super-node cannot be detected. An increase in the WCET upper bound ratio may allow two or more inner nodes of the large super-node to be instrumented instead of the large super-node. The instrumented inner nodes may not cover all BBs

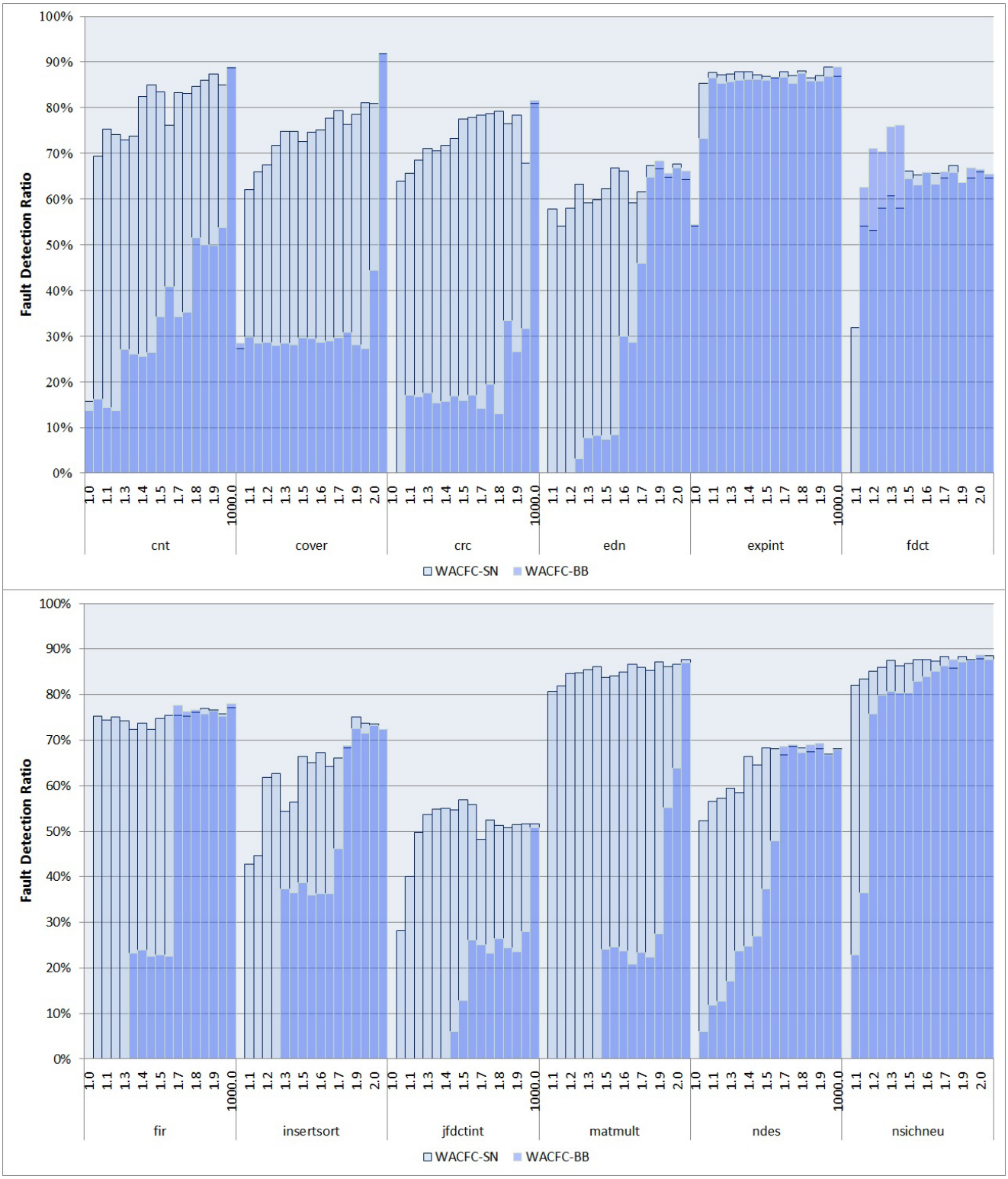


FIGURE 8. Comparisons between Fault Detection Ratios of WACFC-SN and WACFC-BB. All bars are rooted on the horizontal axis and the bars for WACFC-SN (lighter shade) are partially hidden behind the bars for WACFC-BB (heavier shade).

of the large super-node, but invalid control flow between the inner nodes can be detected.

For fault detection ratios, we compare with the algorithm *WACFC-BB (WCET-Aware Control-Flow Checking based on Basic Blocks)* in [5]. Fig. 8 shows comparisons between fault detection ratios for the benchmark programs. As expected, the fault detection ratios for both algorithms increase with increasing WCET upper bound ratio in general.

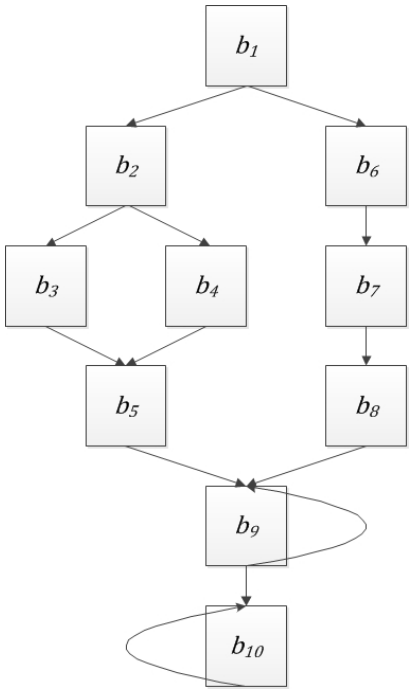


FIGURE 9. CFG of a simple program for the toy example.

WACFC-SN significantly out-performs WACFC-BB on most of the benchmark programs, and achieves similar fault detection ratios on the others (with a few exceptions, e.g. *fdct*). It is not surprising that the fault detection ratio does not increase strictly monotonically with the WCET upper bound ratio, due to the heuristic nature of our algorithm and optimization objective, as well as the inherent randomness of fault injection experiments.

We attribute the better efficiency of WACFC-SN to several reasons:

1) Super-node based CFC provides a larger design space than BB-based CFC, since a BB is a singleton super-node, leading to potentially better results.

2) The optimization objective *CFE Detection Capability* is an improvement over the simplistic optimization objective in [5].

3) For reducing execution time of the current WCEP at each iteration of the algorithm, we formulate and solve an ILP problem (Section III-B) to select the set of super-nodes to be instrumented on the WCEP. This is more effective than the simplistic approach of iteratively selecting and un-instrument the BB on the WCEP that contributes the least to the ACET in [5].

For most benchmark programs that are relatively small, the algorithm running time is several minutes. For larger programs, e.g., *nsichneu* with 4000+ lines of code, the framework finishes within several hours.

V. CONCLUSIONS

In this paper, we present WCET-Aware Control Flow Checking based on Super-Nodes (WACFC-SN), which trades off real-time performance for error resilience by applying CFC instrumentation on a select subset of super-nodes. We evaluate fault detection coverage of WACFC-SN under various WCET upper bounds by fault injection experiments, and demonstrate significant improvements compared with related work.

APPENDIX

In the appendix, we use a toy example to illustrate the Global Super-node Selection algorithm.

Fig. 9 shows the CFG of a program with 10 BBs. There are two loops in the CFG, one containing BB b_9, and the other containing BB b_{10}. The loop counts of both loops are context-dependent: The loop count for the loop containing b_9 is 1 if the left branch is taken, and 10 if the right branch is taken. The loop count for the loop containing b_{10} is 10 if the left branch is taken, and 1 if the right branch is taken. Each BB has the same execution time of 5 (unit is arbitrary; may be cycles); the CFC instrumentation overhead for each BB or super-node is 1, and $\text{WCET}_{\text{ub}} = 85$.

We start from a program with full CFC, where all BBs b_1 to b_{10} are instrumented, so each BB has execution time of 5+1=6. The program WCET is 90 (6*4+6*10+6*1), with both left and right paths as WCEPs. All possible super-nodes, including all singleton super-nodes from (b_1, b_1) to (b_{10}, b_{10}), and non-singleton super-nodes (b_2, b_5), (b_6, b_7), (b_7, b_8), (b_6, b_8), (b_1, b_9), (b_9, b_{10}) and (b_1, b_{10}), are identified, and added to the set $N_{candidate}$ of candidate super-nodes. The set of headers $B_{covered}$ for keeping track of instrumented super-nodes are initialized to contain all BBs.

Suppose the right path b_1, b_6, b_7, b_8, b_9, b_{10} is identified as the current WCEP. The Path-local Super-node Selection algorithm, described in Section III-B, chooses the set of super-nodes (b_1, b_1), (b_6, b_6), (b_7, b_8) and (b_{10}, b_{10}) to be instrumented. The super-nodes instrumented previously on the path are un-instrumented, and all BBs on the current path, i.e., b_1, b_6, b_7, b_8, b_9 and b_{10}, are removed from $B_{covered}$. Then, the selected super-nodes are instrumented, and their headers, i.e., b_1, b_6, b_7 and b_{10}, are added back to $B_{covered}$. $N_{candidate}$ is also updated. Candidate super-nodes that conflict with the selected super-nodes are un-instrumented and removed from $N_{candidate}$. For the current path, singleton super-nodes b_7 and b_8 are removed from $N_{candidate}$, because they are inner nodes of (b_7, b_8). (b_6, b_7) is also removed, because it partially overlaps (b_7, b_8). Note that outer nodes of a selected super-node, e.g., (b_6, b_8), are not removed, in case they may be selected in future. At this point, the execution time of the current path is decreased to 79 $< \text{WCET}_{\text{ub}}$. $B_{covered}$ contains $b_1, b_2, b_3, b_4, b_5, b_6, b_7$ and b_{10}, and $N_{candidate}$ contains (b_1, b_1), (b_2, b_2), (b_3, b_3), (b_4, b_4), (b_5, b_5), (b_6, b_6), (b_9, b_9), (b_{10}, b_{10}), (b_2, b_5), (b_7, b_8), (b_6, b_8), (b_1, b_9), (b_9, b_{10}) and (b_1, b_{10}).

The next WCEP identified by the WCET analysis tool is the left path b_1, b_2, b_3, b_5, b_9, b_{10}, with execution time of 89. Suppose (b_1, b_1), (b_2, b_5) and (b_9, b_9) are selected for instrumentation. All super-nodes on the path are un-instrumented, and b_1, b_2, b_3, b_5 and b_{10} are removed from $B_{covered}$. Then, the selected super-nodes are instrumented,

and their headers b_1, b_2 and b_9 are added back to $B_{covered}$. Inner nodes of (b_2, b_5) are removed from $N_{candidate}$. At this point, the execution time of the current path is $78 < \text{WCET}_{\text{ub}}$. $B_{covered}$ contains b_1, b_2, b_4, b_6, b_7 and b_9, and $N_{candidate}$ contains (b_1, b_1), (b_6, b_6), (b_9, b_9), (b_{10}, b_{10}), (b_2, b_5), (b_7, b_8), (b_6, b_8), (b_1, b_9), (b_9, b_{10}) and (b_1, b_{10}).

However, due to re-instrumentation of (b_9, b_9), execution time of the right path is increased to $88 > \text{WCET}_{\text{ub}}$. The path is identified again as the WCEP, and the same set of super-nodes will be selected for instrumentation. But the re-instrumentation of (b_{10}, b_{10}) will in turn increase the execution time of the left path to $87 > \text{WCET}_{\text{ub}}$. The algorithm would go into an infinite loop, and keep switching between the two paths without making any progress.

With the Deadlock Prevention mechanism, the algorithm checks $N_{candidate}$ and finds that the set stays unchanged after the second time the right path is optimized, which indicates a potential deadlock. So the algorithm adds all candidate super-nodes on the right path to $N_{deadlock}$. The state change counter sc_n for each of the super-nodes n in $N_{deadlock}$ is set to zero, since the super-nodes have not been added to $N_{deadlock}$. At this point, $N_{deadlock}$ contains (b_1, b_1), (b_6, b_6), (b_9, b_9), (b_{10}, b_{10}), (b_7, b_8), (b_6, b_8), (b_1, b_9), (b_9, b_{10}) and (b_1, b_{10}), and $B_{covered}$ contains b_1, b_2, b_4, b_6, b_7 and b_{10}.

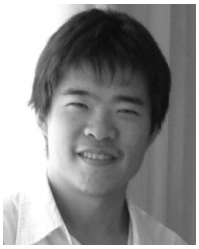
After the second optimization of the right path, re-instrumentation of (b_{10}, b_{10}) causes the left path to be identified as the current WCEP and optimized again. (b_9, b_9) is re-instrumented, and (b_{10}, b_{10}) is un-instrumented. Both the super-nodes experience a selection state change, and $sc_{(b_9,b_9)}$ and $sc_{(b_{10},b_{10})}$ are increased to 1. $N_{candidate}$ remains unchanged again, causing (b_1, b_1), (b_9, b_9), (b_{10}, b_{10}), (b_2, b_5), (b_1, b_9), (b_9, b_{10}) and (b_1, b_{10}) to be added to $N_{deadlock}$. At this point, $N_{deadlock}$ contains (b_1, b_1), (b_6, b_6), (b_9, b_9), (b_{10}, b_{10}), (b_2, b_5), (b_7, b_8), (b_6, b_8), (b_1, b_9), (b_9, b_{10}) and (b_1, b_{10}), and $B_{covered}$ contains b_1, b_2, b_4, b_6, b_7 and b_9.

Once again, the re-instrumentation of (b_9, b_9) causes the right path to be identified as the WCEP and optimized. (b_9, b_9) is un-instrumented, and (b_{10}, b_{10}) is re-instrumented. Both super-nodes experience another selection state change, so $sc_{(b_9,b_9)}$ and $sc_{(b_{10},b_{10})}$ are increased to 2. $N_{candidate}$ remains unchanged again. Furthermore, $N_{deadlock}$ also converges to a fixed set, which indicates that the algorithm is trapped in a finite set of paths and a deadlock has probably occurred. So the candidate super-node in $N_{deadlock}$ that has experienced the most selection state changes, i.e., (b_9, b_9) or (b_{10}, b_{10}), is un-instrumented and removed from $N_{candidate}$, and its header is removed from $B_{covered}$. Suppose (b_9, b_9) is un-instrumented and removed from $N_{candidate}$. The deadlock is now broken, so $N_{deadlock}$ is cleared. At this point, the WCET of the right path is $79 < \text{WCET}_{\text{ub}} = 85$.

Re-instrumentation of (b_{10}, b_{10}) again causes the left path to be optimized. But this time, (b_9, b_9) is not in $N_{candidate}$ and will not be instrumented. The algorithm un-instruments (b_{10}, b_{10}) to decrease the execution time of the path, and removes b_{10} from $B_{covered}$. At this point, the WCET of the left path is $77 < \text{WCET}_{\text{ub}}$. $B_{covered}$ contains b_1, b_2, b_4, b_6, and b_7. Finally, super-nodes to be instrumented can be identified by searching for the innermost candidate super-nodes that contain at least one BB in $B_{covered}$. For b_1, b_6 and b_7, the innermost covering candidate super-nodes are (b_1, b_1), (b_6, b_6), and (b_7, b_8), respectively. For b_2 and b_4, the innermost covering candidate super-node is (b_2, b_5). So (b_1, b_1), (b_6, b_6), (b_2, b_5) and (b_7, b_8) are finally instrumented.

REFERENCES

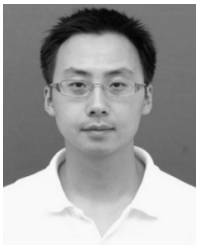
[1] R. Shimmin *et al.*, "The successful PhoneSat wifi experiment on the Soarex-8 flight," in *Proc. Aerosp. Conf.*, 2016, pp. 1–9.
[2] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*, 1st ed. New York, NY, USA: Springer, 2006.
[3] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," *IEEE Trans. Rel.*, vol. 51, no. 1, pp. 111–122, Mar. 2002.
[4] M. Lv, Z. Gu, N. Guan, Q. Deng, and G. Yu, "Performance comparison of techniques on static path analysis of WCET," in *Proc. EUC*, 2008, pp. 104–111.
[5] Z. Gu, C. Wang, M. Zhang, and Z. Wu, "WCET-aware partial control-flow checking for resource-constrained real-time embedded systems," *IEEE Trans. Ind. Electron.*, vol. 61, no. 10, pp. 5652–5661, Oct. 2014.
[6] D. Zhu, "Reliability-aware dynamic energy management in dependable embedded real-time systems," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, Dec. 2010, Art. no. 26.
[7] Z. Li, F. Lockom, and S. Ren, "Maintaining real-time application timing similarity for defect-tolerant NoC-based many-core systems," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 2s, Jan. 2014, Art. no. 64.
[8] Z. Li, L. Wang, S. Li, S. Ren, and G. Quan, "Reliability guaranteed energy-aware frame-based task set execution strategy for hard real-time systems," *J. Syst. Softw.*, vol. 86, no. 12, pp. 3060–3070,Dec. 2013.
[9] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," *IEEE Trans. Comput.-Aided Design Integr.*, vol. 16, no. 12, pp. 1477–1487, Dec. 1997.
[10] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, "WCET centric data allocation to scratchpad memory," in *Proc. RTSS*, 2005, pp. 223–232.
[11] Z. Wang, Z. Gu, and Z. Shao, "WCET-aware energy-efficient data allocation on scratchpad memory for real-time embedded systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 11, pp. 2700–2704, Nov. 2015.
[12] Z. Wang, Z. Gu, M. Yao, and Z. Shao, "Endurance-aware allocation of data variables on NVM-based scratchpad memory in real-time embedded systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 34, no. 10, pp. 1600–1612, Oct. 2015.
[13] J. L. Hennessy and D. A. Patterson, *Computer Architecture—A Quantitative Approach*, 5th ed. San Mateo, CA, USA: Morgan Kaufmann, 2012.
[14] R. Vemu and J. A. Abraham, "Budget-dependent control-flow error detection," in *Proc. IOLTS*, Jul. 2008, pp. 73–78.
[15] D. Khudia and S. Mahlke, "Low cost control flow protection using abstract control signatures," in *Proc. LCTES*, 2013, pp. 3–12.
[16] G. Nazarian, R. Nane, and G. Gaydadjiev, "Low-cost software control-flow error recovery," in *Proc. DSD*, 2015, pp. 510–517.
[17] M. Lee, P. Tirumalai, and T.-F. Ngai, "Software pipelining and superblock scheduling: Compilation techniques for VLIW machines," in *Proc. HICSS*, vol. 1, 1993, pp. 202–213.
[18] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," in *Proc. WCET Workshop*, 2010, pp. 136–146.
[19] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. CGO*, 2004, pp. 75–86.



MING ZHANG is currently pursuing the Ph.D. degree with the College of Computer Science, Zhejiang University. His research interests include embedded systems and neuromorphic computing.

ZONGHUA GU received the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 2004. He is currently an Associate Professor with the College of Computer Science, Zhejiang University. His research interests are embedded and cyber-physical systems.

HONG LI received the Ph.D. degree from the Department of Computer Science, Zhejiang University, in 2010. Her research interests include embedded real-time systems, intelligent systems, and cyborg OS.



NENGGAN ZHENG received the Ph.D. degree from Zhejiang University, Hangzhou, China, in 2009, where he is currently an Associate Professor in computer science with the Academy for Advanced Studies. His current research interests include artificial intelligence, embedded systems, and brain–computer interface.