PREPRINT: "H. Kashif, J. J. Thomas, H. Patel, and S. Fischmeister, "Static slack-based instrumentation of programs," in 20th IEEE Conference on Emerging Technologies \\& Factory Automation, ETFA, 2015, pp. 1-8."

# Static Slack-Based Instrumentation of Programs

Hany Kashif, Johnson Thomas, Hiren Patel, Sebastian Fischmeister
Dept. of Electrical and Computer Engineering
University of Waterloo, Canada
{hkashif, j22thoma, hdpatel, sfischme}@uwaterloo.ca

***Abstract*—Real-time embedded programs are time sensitive and, to trace such programs, the instrumentation mechanism must honor the programs' timing constraints. We present a time-aware instrumentation technique that injects program code with slack-based conditional instrumentation. The central idea is to execute instrumentation code only when its execution does not increase the worst-case execution time beyond a program's deadline. This occurs at run-time. Unlike previous efforts, this work allows instrumenting on the path that results in the worst-case execution time of the program. We propose a software, and a hardware method of allowing for slack-based conditional instrumentation. We evaluate and compare these two alternatives using a common benchmark suite for real-time systems. Our results show that, on average, the two proposed methods achieve 57% and 80% instrumentation coverage, respectively, compared to only a 3% coverage by previous work.**

## I. Introduction

Testing and validation is an integral component of the software design process that ensures that the software under test is correct. However, providing such software correctness guarantees is both difficult [1] and expensive [2]. The process of diagnosis typically employs program tracing techniques.This is done by instrumenting the program to trace and/or monitor the program state. The designer inspects the trace after execution to identify potentially erroneous state information.

The above-mentioned issues are further aggravated for hard real-time systems which demand guarantees on temporal behaviors in addition to functional correctness. Consequently, any instrumentation to the original program code for the purpose of program tracing may affect the temporal behaviors of the program. Software-based instrumentation approaches [3], [4] insert tracing and monitoring code into the original program code. Typically, the more tracing code the program executes during the run, the more the perturbation in temporal behaviors. Dynamic instrumentation approaches [5], [6] modify the binary at run time and thus cause highly non-deterministic timing behavior. While DIME [7] limits the dynamic instrumentation overhead to a pre-specified budget, overshoots beyond the timing budget can occasionally occur making DIME more suited for soft real-time applications. Hardware-supported tracing [8], [9] use special hardware interfaces to stream data off chip. Special hardware is costly and is not available for all processors. This method as well can cause significant perturbation [10].

Recent work in time-aware instrumentation [11], [12], [13] investigates mechanisms to instrument programs without affecting their specified timing constraints, and functional behavior. The underlying idea is to instrument programs only in places where it leaves the original behavior unaffected and still obeys all timing constraints. Figure 1 illustrates the effect of time-aware instrumentation on the execution time profile. As we instrument a time-sensitive program using the ideas of time-aware instrumentation, we shift the execution time profile closer to the program's deadline (end of the time budget). While the work in [11], [12], [13] exudes promise, it has a central restriction that portions of the program on the worst-case path (WCP) cannot be instrumented. The repercussions of this restriction are apparent in the results of a case study of a one-laptop per child (OLPC) keyboard controller reported in [11]. The OLPC case study showed that the WCP shared more than 25% of its code with other paths in the program. Hence, large portions of the program cannot be instrumented because adding instrumentation to these portions would increase the execution time on the WCP.



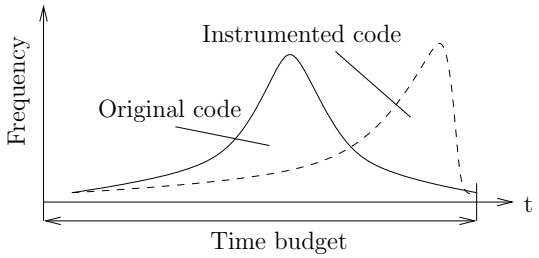



Fig. 1: The underlying idea of time-aware instrumentation [11]

In this work, we propose a slack-based conditional instrumentation (SCI) technique for debugging hard real-time programs. SCI preserves functional behavior, and temporal constraints of the original program while allowing the instrumentation of variables on the WCP. First, we introduce SCI which allows the instrumented code to execute only when there is sufficient slack in the program. Then, we address the challenge of selecting points in the program code to insert such SCI. Finally, we use a purely software technique to implement SCI and compare it against a technique that extends the processor with instructions to perform the conditional check. Notice that both of these techniques check for slack at run-time, and execute the instrumented code only if at run-time there exists sufficient slack.

We would like to point out that SCI is *not* a method for WCET analysis. WCET is only an input to our instrumentation technique. We apply our instrumentation method for data tracing in our examples. Potential application areas include tracing, logging, and runtime verification.

## II. The Underlying Concepts

We refer to conditional instrumentation code points, which are conditionally executed at run-time based on available run-time slack, as conditional points (CPs). We also refer to

instrumentation code points used in previous work [11], [12], which are always executed at run time, as instrumentation points (IPs). We assume that we can reliably estimate the WCET of instrumentation code points. Hence, there are two types of instrumentation points that we can insert into a program: CPs and IPs.

We borrow the abstract model presented in [11], and augment it to include our proposed approach for SCI. The abstract model represents the source program as an extended control-flow graph (CFG). A basic block is a portion of source code of the program with one entry point and one exit point. We augment the definition of a vertex with an associated type as shown in Definition 1. Using this definition of a vertex, we define the extended CFG as shown in Definition 2. We call this abstract model a one state-change CFG (OSCCFG).

**Definition 1** (Vertex)**.** *A vertex is a basic block with at most one assignment to the same variable. We represent a vertex as a tuple $v = (i, t)$ where $i \in \mathbb{N}$ is a unique identifier, and $t \in \{$None,IP,CP$\}$ is an instrumentation type.*

The unique identifier allows us to distinguish and reference vertices. A vertex with the *None* instrumentation type is the default for all vertices. *IP* indicates that an instrumentation code point (IP) is added to that vertex, and *CP* denotes a conditional instrumentation code point (CP). Only one instrumentation point, either an IP or a CP, can be added to a vertex. Note that a vertex is a basic block of the program with the additional requirement that each basic block modifies any variable at most once within it. A traditional CFG contains vertices with multiple state changes to the same variables via assignments within a vertex. We split such vertices into multiple vertices with only one state change to the same variable in each vertex, and construct edges between them. In this modified CFG, adding one instrumentation point to a vertex would be sufficient to capture all state changes of the modified variables in this vertex. This modification facilitates the traversal and instrumentation of the input program.

**Definition 2** (One state-change CFG)**.** *A one state-change control-flow graph is a directed graph $\mathcal{G} := \langle V, E, v_s, v_x \rangle$ where $V$ is the set of vertices, $E \subseteq V \times V$ is the set of edges that represent flow of control, and $v_s, v_x \in V$ are unique start and exit vertices, respectively.*

A path of an OSCCFG $\mathcal{G}$ describes a traversal of the graph as shown in Definition 3. We denote the set of all paths from $v_s$ to $v_x$ as $\mathbb{P}_{v_s,v_x}$. We identify the WCP as the path with the largest WCET estimate [14]. To extract the sequence of vertices from a path $p_{v_s,v_x}$ we employ the helper function *vertices* : $\mathbb{P}_{v_s,v_x} \rightarrow V^{[]}$. Notice that we superscript domains with [] to denote a sequence and $\{\}$ for a set. We give an example of these definitions in Section III.

**Definition 3** (Path)**.** *A path $p_{v_s,v_d}$ from source vertex $v_s$ to destination $v_d$ in an OSCCFG $\mathcal{G}$ is a sequence of vertices $\langle v_s = v_1, v_2, \ldots, v_{n-1}, v_n = v_d \rangle$ with $n \in \mathbb{N}$ being the number of vertices forming the path.*

Figure 2 shows an example execution time profile of a program. The WCET of a program is an upper-bound on the execution time of any path in the program [14]. The difference between the WCET and the actual execution time of any program instance is commonly called run-time slack [15]. Note that the actual execution time is a run-time characteristic. SCI uses this run-time slack to execute instrumentation code. The static time window, $\alpha$, is the difference between the program's WCET and deadline such that $\alpha = Deadline - WCET$.

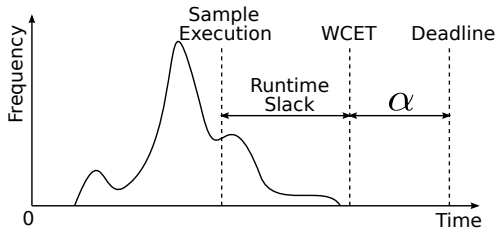



Fig. 2: Example of a program's execution time variation

We typically insert IPs at vertices that lie on paths other than the WCP. For instrumenting vertices on the WCP, we use CPs. It might occur that after instrumentation, the WCP changes. We discover this through rerunning the WCET analysis after instrumentation. If the WCP changes, we will convert all instrumented vertices on the new WCP from IPs to CPs.

CPs check whether there is sufficient run-time slack available to execute the instrumentation code, and if there is, then the program will execute the instrumentation code; otherwise, the program will skip it. We call the portion of code in the CP that checks whether sufficient run-time slack is available to execute the instrumentation code as the overhead of the CP. This code portion (overhead of a CP) is always executed.

Adding CPs on the WCP may lead to an increase in the WCET because of the additional overhead. However, WCETs are typically lower than the application deadline. This means that perturbations in the WCETs are acceptable as long as they are less than or equal to the static slack time window, $\alpha$, specified by the application. This allows us to absorb small increases in the WCET, and still ensure that the temporal deadlines of the program are correct.

## III. Illustrative Example of SCI

We illustrate SCI with the example shown in Listing 1. We annotate Listing 1 with labels A, B, C, D, E, and F that identify vertices for its OSCCFG $\mathcal{G}$ shown in Figure 3.

There are two paths in $\mathcal{G}$: $p_1 = \langle A, B, E, F \rangle$, and $p_2 = \langle A, C, D, E, F \rangle$. Let us assume that path $p_2$ is the WCP in this example, which we show as shaded vertices. We want to trace all state changes to variables $x$ and $y$. To trace all state changes of $x$ and $y$, in this example, we compare naive instrumentation (IPs on WCP), CPs on WCP (ALLCP), and minimal CPs on WCP (MINCP). This comparison highlights the problem with using IPs on the WCP, and motivates the minimization of CPs on the WCP. Consider the following *hypothetical* setting. Assume that each basic block in $\mathcal{G}$ has a WCET of 2 time units and an actual execution time (ET) of 1.5 time units. Let the cost of recording and retrieving the current state of the program be 0.8 time units and recording a single variable to a buffer be 0.08 time units. Let the cost of checking whether sufficient run-time slack exists be 0.1 time

```
1  A: x++;
       if ( x > 10 ) {
3  C:   c = z;
       z = y;
5      y = c;
   D:  c = x;
7    } else {
   B:  z++;
9    }
   E: z++;
11   y += z;
   F: z = c;
13   x++;
```

Listing 1: C program without instrumentation

units. For example, to check whether sufficient run-time slack exists and if so, record a variable $x$ to the buffer, the cost incurred would be 0.1+0.8+0.08=0.98 time units. Assume that the deadline assigned to the program by the developer is 11 time units, hence, $\alpha = 11-2*5 = 1$. Note that instrumentation points are executed at the end of basic blocks but before any branches are executed.

- **Naive: All IPs on WCP.** Since $A$, $C$, $E$, and $F$ modify either $x$ or $y$, we add IPs to these basic blocks. Using the above setting and upon execution of the program, this results in an execution time of 7.5+(0.88)*4 = 11.02 time units, which exceeds the deadline (11).
- **ALLCP: All CPs on WCP.** We add CPs to basic blocks $A$, $C$, $E$, and $F$, as shown in Figure 4, which also shows the buildup of run-time slack as the program executes. If a basic block annotated with CP has sufficient run-time slack to execute the instrumentation, then the instrumentation block will execute, hence reducing the run-time slack by 0.98, otherwise the run-time slack is reduced by 0.1. $A$ and $C$ have insufficient run-time slack to execute the instrumentation code to record $x$ and $y$, respectively. At $A$ and $C$, after the basic block executes, the run-time slack increases by 0.5 but decreases by 0.1 to check for run-time slack at the CP. $E$ and $F$ have sufficient run-time slack to execute instrumentation code, and consume 0.98 time units each to record $y$ and $x$, respectively.
- **MINCP: Minimal CPs on WCP.** We delay recording of variables $x$ and $y$ till just before any of them gets overwritten. We add CPs only at basic blocks $D$ and $F$ which record both variables $x$ and $y$ as shown in Figure 5. The cost of recording both $x$ and $y$ at a basic block, including checking for run-time slack, is $0.8 + 0.08 * 2 + 0.1 = 1.06$. It can be seen that sufficient run-time slack is available at $D$ and $F$ to record both variables.

One can infer from the above example the following conclusions. The addition of IPs on the WCP could lead to a violation of the deadline. With addition of CPs on the WCP, there is a higher chance of execution of the instrumentation code if placed at a vertex that is closer to the exit vertex. This is because run-time slack builds up as basic blocks are executed during program execution. SCI exploits this by minimizing CPs and delaying the tracing of variables (Section IV). Lastly, with every CP introduced in the program, there is a mandatory cost of checking whether sufficient run-time slack exists at each CP. If the cost of checking for run-time slack is high enough for the program to miss its deadline, then one will have to selectively insert CPs in the program to record the maximal number of state changes of variables of interest (Section V).

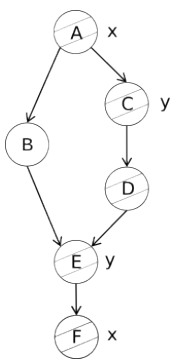



Fig. 3: Original

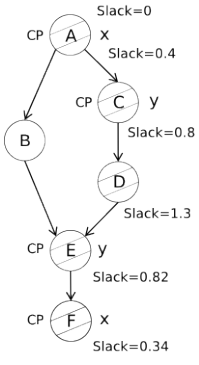



Fig. 4: All CPs

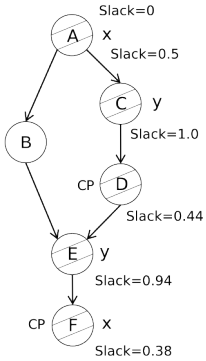



Fig. 5: Min. CPs

## IV. Minimization of CPs on The WCP (MINCP)

We describe the approach which minimizes the number of CPs that capture all possible state changes of the variables that we want to trace on the WCP. MINCP maintains a set of modified variables. We delay capturing the state change of this set of modified variables till just before any of the variables gets overwritten. When we delay capturing a state change, the program executes more code, which probably leads to gaining more run-time slack and, thus, increases the likelihood of the execution of CPs as illustrated in Section III. We denote the set *traceVars* $\subseteq$ *VARS* as the variables we want to trace, where *VARS* is the set of all variables in the program. MINCP extracts a set of vertices to instrument with CPs to capture all state changes of the variables in *traceVars*.

We illustrate MINCP by revisiting the OSCCFG shown in Figure 5. Path $p_{A,F} = \langle A, C, D, E, F \rangle$ is the WCP, which we show as shaded vertices. MINCP adds a CP at vertex $D$ to instrument variables $x$ and $y$ before $y$ is modified in vertex $E$. It also adds a CP at vertex $F$ to instrument the state changes of $y$ and $x$ at vertices $E$ and $F$, respectively.

We show the MINCP algorithm in Function 1. It takes as input: *traceVars* and the WCP $p_{v_s,v_x}$. The output is $V^{\{\}}$, the set of vertices to be instrumented. We introduce several helper functions in describing this algorithm. To extract the set of variables being assigned new values in a vertex, we use function *modifiedVars*$(v) : V \rightarrow var^{\{\}}$. We use the function $predecessor(v) : V \rightarrow V$ to extract the vertex that has an incident edge on $v$ on the WCP. We use functions $scopeBegin : V \rightarrow \{true, false\}$ and $scopeEnd : V \rightarrow \{true, false\}$ to identify the beginning and end of scopes, respectively. A scope corresponds to a loop on the WCP and can be identified by static analysis [16] (also applies to continue and break statements). Note that if a vertex marks the beginning of multiple scopes, it will be split into multiple vertices such that each new vertex marks the beginning of only one scope. Function $getScopeVars : V \rightarrow var^{\{\}}$ extracts the set of variables that are modified within a scope where the input argument to $getScopeVars$ is the beginning of the

scope. The stack operations *push* and *pop* are used to push/pop an element into/from a stack, respectively.

**Function 1** Minimization of CPs on WCP

**Input:** *traceVars*, $p_{v_s,v_x}$
**Output:** $V^{\{\}}$

```
    Let M ⇐ ∅ be the set of variables being monitored
 2: Let I ⇐ ∅ be the set of instrumented vertices
    Let S be an empty stack
 4:
    for v ∈ vertices(p_{v_s,v_x}) do
 6:    modVars ← modifiedVars(v) ∩ traceVars

 8:    if scopeBegin(v) then
          if getScopeVars(v) ∩ M ≠ ∅ then
10:           I ← I ∪ {(predecessor(v), M)}
              M ← ∅
12:       end if
          push(S, M)
14:   end if

16:   if M ∩ modVars ≠ ∅ then
          I ← I ∪ {(predecessor(v), M)}
18:       M ← ∅
      end if
20:   M ← M ∪ modVars

22:   if scopeEnd(v) then
          if M ≠ ∅ then
24:           I ← I ∪ {v, M}
          end if
26:       M ← pop(S)
      end if
28: end for

30: if M ≠ ∅ then
      I ← I ∪ {v_x, M}
32: end if
    return I
```

The core idea of the algorithm is to delay the recording of a variable change until the point where at least one of the variables of interest gets overwritten. Function 1 iterates through the vertices on the WCP $p_{v_s,v_x}$ in order. While iterating through the vertices, set $M$ holds the variables of interest that have been modified without any of the variables being overwritten. Set $I$ holds the set of vertices to be conditionally instrumented along with the variables to instrument at each vertex. For each vertex $v$, the algorithm extracts the set $modVars$ which is the set of variables of interest that vertex $v$ modifies (line 6). First, the algorithm checks whether vertex $v$ begins a new scope (line 8). If vertex $v$ begins a new scope and this scope modifies any of the variables in set $M$ (lines 8-9), then the algorithm will choose to instrument all variables in $M$ before entering the scope (lines 10-11), i.e., at the vertex preceding the beginning of the scope (excluding the loop's back-edge). After a new scope starts, the set $M$ is pushed into a stack $S$ (line 13). If a vertex modifies any of the variables in the set $M$, then the algorithm will choose to instrument variables in $M$ at the preceding vertex (lines 16-19). Afterwards, set $M$ will be updated with the set of modified variables at vertex $v$ (line 20). If a vertex $v$ marks the end of a scope, then the vertex $v$ will be instrumented with the variables in set $M$ (if any) and set $M$ will be popped from the stack $S$ to restore the set of modified variables before the scope started (lines 22-27). If the exit vertex is reached and the set $M$ is not empty, then the exit vertex will be instrumented with the variables in set $M$ (lines 30-32). Finally, we iterate through the set $V^{\{\}}$ obtained as output from Function 1 and add CPs to these vertices to record the specific variables associated with each vertex. All helper functions but $getScopeVars$ have a constant time complexity with respect to the number of vertices on the WCP. Function $getScopeVars$ loops on all vertices within a scope and, thus, has a linear complexity. The complexity of the algorithm is, therefore, quadratic in the number of vertices on the WCP.

## V. Constrained Minimization of CPs (C-MINCP)

We augment MINCP to consider a constraint on the increase in the WCET caused by instrumenting the WCP. We attempt to maximize coverage by minimizing the number of CPs to capture the maximum number of state changes of variables in *traceVars* given a time budget.

We use Function 1 to get a minimal number of CPs required to trace all the state changes of variables of interest (*traceVars*). Then, we select a subset of these CPs such that we trace the maximum number of variables of interest, and the overhead incurred by the CPs is within the static window $\alpha$. We can describe the problem of selecting a subset of CPs using Equation 1.

$$\text{Max} \sum_{i=1}^{n} b_i * (\textit{varsInCP}_i * \textit{frequency}_i)$$
$$\text{subject to} \sum_{i=1}^{n} b_i * (\textit{overhead}_i * \textit{frequency}_i) \leq \alpha \quad (1)$$

Here, $\textit{varsInCP}_i$ is the number of variables monitored in the CP $i$, $\textit{frequency}_i$ is the number of times CP $i$ is attempted, and $\textit{overhead}_i$ is the overhead of CP $i$. The total number of CPs, $n$, is obtained from Function 1, and $b_i$ is the binary integer programming (BIP) variable. Notice that we do not include the execution time incurred by the instrumentation code because at run-time we determine whether we have sufficient run-time slack to execute the instructions monitoring the variables. Note also that the WCET analysis tool determines $\textit{frequency}_i$. It is important to clarify that $\textit{frequency}_i$ is *not* the number of times the instrumentation code inside CP $i$ executes, but rather the frequency of executing the conditional check of the CP, i.e. the number of times the CP is attempted. Although using the value supplied by the WCET analysis tool is pessimistic, our goal is to make sure that we honor the program's timing constraints.

Solving our problem for finding a subset of CPs to create is NP-Complete (the problem is polynomially reducible to the binary knapsack problem and the problem $\in$ NP as it is verifiable in polynomial time). We use standard BIP tools to solve the optimization problem and instrument the resulting vertices picked by the BIP solver.

## VI. Implementation Approaches

We experiment with two implementations of SCI: software-based and hardware-based.

### A. Software Implementation

The software implementation uses function calls in the program to extract cycle counter values. Listing 2 shows a simple example of a software implementation of SCI. Functions *func_a* and *func_b* have WCETs of $wcet_a$ and $wcet_b$, respectively. Assume the WCET of all instrumentation code at labels B, C, and D is $wcet_{c_1}$ and at labels E and F is $wcet_{c_2}$.

```
1  int main(void) {
   A:    globalTime = getTime() + wcet_a;
3        func_a();
   B:    if (globalTime − getTime() >= wcet_c1) {
5  C:          // Instrumentation Code
                .....
7        }
   D:    globalTime += wcet_b;
9        func_b();
   E:    if (globalTime − getTime() >= wcet_c2) {
11 F:          // Instrumentation Code
                .....
13       }
   }
```

Listing 2: A software implementation of SCI

When the program executes, it sets variable *globalTime* at label A, to the time at which function *func_a* will finish execution in the worst-case. After *func_a* completes, the instrumentation code compares *globalTime* to the current time to check whether there is sufficient run-time slack to execute the instrumentation code of *func_a*. The instrumentation code then updates *globalTime* at label D to hold the time at which function *func_b* finishes in the worst-case. The same check for instrumentation is repeated after *func_b*.

The function *getTime* is *not* an OS function call but rather an instruction that reads a dedicated free running hardware timer on the chosen processor. Hardware timers exist in the processors used for embedded systems and are either memory- or register-mapped timers.

### B. Hardware Implementation

Our hardware implementation requires extensions to the instruction-set architecture (ISA).

*1) Hardware Extensions:* We extend a cycle-accurate ARMv5 architecture platform with a 32-bit count-down timer, and we extend its ISA with two instructions. We introduce the set timer *stt* instruction, and a check time *chk* instruction. Figure 6 shows the *stt* and *chk* instruction encodings. The *stt* instruction has a single 16-bit immediate operand ⟨*timH:timL*⟩ while the *chk* instruction has two 8-bit immediate operands: ⟨*slk*⟩ and ⟨*raddrL:raddrH*⟩. Every clock cycle, the 32-bit timer will decrement its value by one if it is greater than zero. The *stt* instruction adds its 16-bit operand, ⟨*timH:timL*⟩, to the value already in the timer. The *chk* instruction compares its first 8-bit operand ⟨*slk*⟩ to the value of the timer, and if the first operand value is greater than the timer value then the processor will branch past the number of instructions specified in the second 8-bit operand of the instruction, ⟨*raddrL:raddrH*⟩. Otherwise, the code will execute normally without branches.

We incorporate the *stt* and *chk* instructions into the five-stage pipelined architecture consisting of *Fetch*, *Decode*, *Execute*, *Memory*, and *Writeback* stages. In the *Fetch* stage, the processor fetches the instruction at the address in the program counter and increments the program counter by four. In the *Decode stage*, the processor decodes the instructions and reads the value of the 32-bit timer in the case of *stt* or *chk*. In the *Execute stage* of the *stt* instruction, the ALU adds the value of the 16-bit operand to the timer value and the processor writes back the result to the timer in the *Writeback stage*. In the case of executing the *chk* instruction, the ALU subtracts the first 8-bit operand from the timer value in the *Execute* stage. In the same stage, the branch logic shifts the second 8-bit operand two bits to the left and adds the result to the new program counter value from the *Fetch stage*. If the result of the ALU operation is negative, then a MUX will set the program counter to the output of the branch logic in the next *Fetch* stage, i.e., a branch will occur. Otherwise, the MUX output will set the program counter to the incremented value of the program counter from the previous *Fetch* stage.

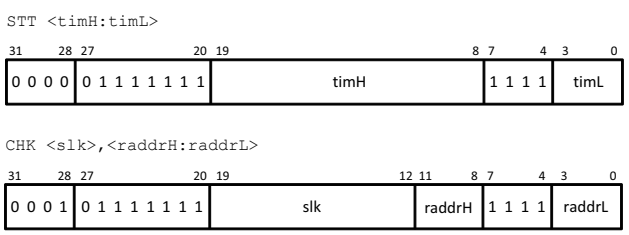



Fig. 6: Instruction encodings of *stt* and *chk* instructions

*2) Functional Operation:* Listing 3, a rework of Listing 2, illustrates the use of *stt* and *chk* instructions in SCI. Functions *func_a* and *func_b* have WCETs of $wcet_a$ and $wcet_b$ cycles, respectively. The instrumentation code and the *chk* and *stt* instructions at labels B, C, and D have a WCET $wcet_{c_1}$ cycles. The instruction count in the instrumentation code at label C is $instr_{c_1}$. The *chk* instruction and instrumentation code at labels E and F have a WCET $wcet_{c_2}$ cycles. The instruction count in the instrumentation code at label F is $instr_{c_2}$.

```
   int main(void){
2  A:    asm(" stt  wcet_a");
         func_a();
4  B:    asm("chk wcet_c1,instr_c1");
   C:    // Instrumentation Code
6            .....
   D:    asm(" stt  wcet_b");
8        func_b();
   E:    asm("chk wcet_c2,instr_c2");
10 F:    // Instrumentation Code
            .....
12 }
```

Listing 3: SCI using *stt* and *chk* instructions

For one execution of the example shown, functions *func_a* and *func_b* will have an actual execution time of $exec_a$ and $exec_b$ cycles, respectively. The *stt* instruction at label A sets the timer to $wcet_a$ (assuming the timer is initialized to zero at the start of program execution). The *chk* instruction at label B compares the timer ($wcet_a - exec_a$) to $wcet_{c_1}$ which is the time needed to instrument function *func_a*. If the timer is greater than or equal to the instrumentation time, the processor will execute the instrumentation code. Otherwise, the processor will branch forward $instr_{c_1}$ instructions, past

the instrumentation instructions to function *func_b*. The next pair of *stt* and *chk* instructions operate similarly for function *func_b*. These instructions make use of accumulated run-time slack. If this slack is insufficient to execute a CP, the timer will carry forward the run-time slack to use at the next CP.

## VII. Experimentation

We implement a fully automated tool for SCI. We use the Unisim cycle-accurate simulator [17] for our implementations and ISA extensions. In Unisim, for the software implementation, we implemented *getTime* as a single instruction that reads the simulator's timestamp. We use the Unisim's default configurations including the latencies for register and memory accesses. Our tool extracts the OSCCFG from the program's C source code. The tool invokes RapiTime v2.4 [18] to extract the WCET of each basic block and to find the program's WCP. The tool generates and inserts the software or hardware instrumentation points. Finally, the tool cross compiles the instrumented program for Unisim, runs the cycle-level simulation, and extracts the logged trace data.

We experiment with the SNU real-time benchmark suite [19], which contains 17 C benchmarks that implement numeric and DSP algorithms. They have 117 lines of code and 34 basic blocks on average. We compare our approaches with the technique proposed by Fischmeister et al. [11], which we refer to as *previous work*. We trace all variables, except function arguments, constants, and loop counters, by logging them to dedicated memory buffers.

We quantitatively assess SCI using the following metrics:

- **Instrumentation coverage:** Instrumentation coverage along an execution path shows the ratio of traced variable assignments to those that a developer desires to trace. It is the probability that instrumentation captures variable assignments before their re-assignment.
- **WCET Overhead:** In the worst-case scenario, an instrumented program executes its WCP such that there is insufficient run-time slack to execute any CPs on that path. However, there is an increase in the program's WCET because CPs add overhead to the WCP (including conditional checks and dynamic slack computation).
- **Code Size Overhead:** Every instrumentation point adds extra code to the original source code. The less the code size overhead, the more effective the instrumentation approach is in utilizing code space for instrumentation.

Figure 7 presents the instrumentation coverage of the software and hardware implementations of ALLCP and MINCP against previous work. Recall that ALLCP inserts a CP on the WCP at each vertex where a variable of interest is assigned without minimization (Section III). Previous work is only able to instrument six benchmarks. SCI is able to trace 16 and 13 benchmarks using the hardware and software implementations, respectively. None of the instrumentation techniques was able to instrument the *fibcall* benchmark. This means that there is insufficient run-time slack at all CPs.

The hardware implementation clearly increases the instrumentation coverage versus software. For the hardware implementation, MINCP has higher coverage than ALLCP. However, the *jfdctint* benchmark violates this rule. This shows that minimizing CPs does not necessarily lead to a higher instrumentation coverage. The reason is that although the run-time slack might be sufficient to execute a small instrumentation code, it might be insufficient to execute larger instrumentation code where the smaller one is merged. This also explains why for the *sqrt* and *crc* benchmarks, using the software implementation, ALLCP has higher coverage than MINCP.

We also compare the instrumentation coverage for the software implementation of ALLCP against previous work in Table I. This table shows the mean value, the 95% confidence interval, and the standard error of mean. Even with conservative estimates, the software implementation is at least one order of magnitude better than previous work.

TABLE I: Instrumentation coverage

| Instrumentation | Mean | 95% CI | SEM |
|---|---|---|---|
| Previous Work | 0.026 | 0.040 | 0.019 |
| ALLCP- Software | 0.569 | 0.220 | 0.104 |

We also analyze the different implementations of ALLCP and MINCP for the WCET and code size overheads (values omitted due to space constraints). The values of the WCET and code size overheads for the software approach are relatively high as compared to the WCET and the code size of the original programs. Although previous work leaves the WCET of the WCP unchanged and has minimal increase in code size, it also has least coverage of the proposed approaches.

We compare the overheads of our approaches to the software implementation of ALLCP. The code size overhead of MINCP software, ALLCP hardware, and MINCP hardware have values of 92.4%, 31.9% and 29.7%, on average. In the worst-case, the hardware methods minimize the code size overhead 2.3 times as compared to the software methods. Moreover, the hardware implementation decreases the WCET overhead. On average, compared to the WCET overhead of ALLCP software, MINCP software, ALLCP hardware, and MINCP hardware have values of 89.4%, 21.0%, and 22.7%, respectively.

To experiment with the software and hardware implementations of C-MINCP, we present results for the *minver* program which performs a 3x3 matrix inversion. Notice that to see the effect of C-MINCP in choosing CPs, we need to synthetically generate versions of the program each with a different time budget. To collect results, we (1) calculate the overhead and the number of traced variables at each CP, (2) measure the frequency of execution of each CP, (3) increment the static slack $\alpha$ by one cycle (starting by 0), (4) run the algorithm given in Section V to instrument the program, (5) compile and cycle-level simulate the program, and (6) repeat steps 3, 4, and 5 until we reach the maximum instrumentation coverage.

Figure 8a shows the instrumentation coverage of the *minver* benchmark as the static slack $\alpha$ increases. We observe that the instrumentation coverage increases as $\alpha$ increases as expected. This is because more CPs can be inserted in the code. Notice that increasing $\alpha$ after all CPs are inserted, increases the coverage because it adds more initial slack to the program which executes more CPs. It is apparent from the figure that at some points increasing $\alpha$ leads to less coverage. The reason is that increasing $\alpha$ might lead to inserting a CP instead of a few others because the former traces more variables. Typically, this

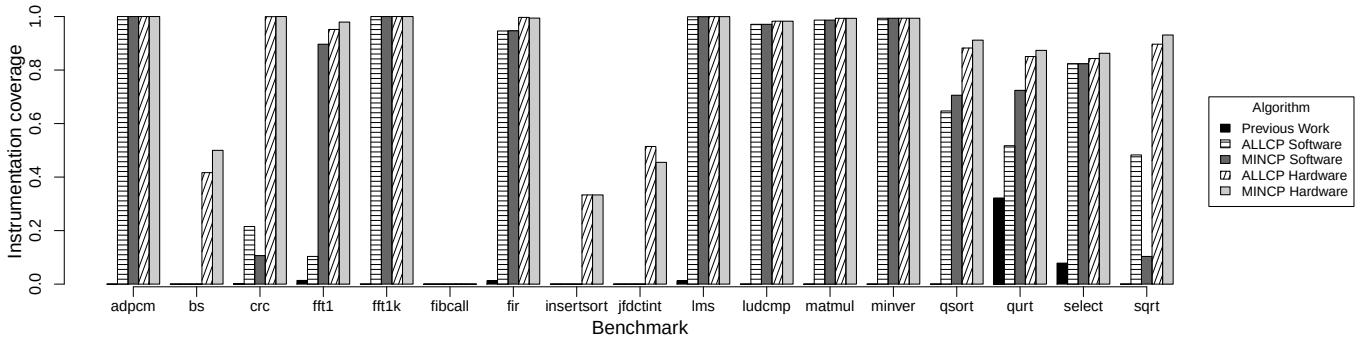



Fig. 7: Instrumentation coverage of software and hardware implementations of ALLCP and MINCP

should lead to higher coverage, however, the run-time slack might be insufficient for the execution of the larger CP.

Figures 8b and 8c present the variation in code size and WCET overheads as $\alpha$ increases, respectively. Generally, code size and WCET overheads increase as $\alpha$ increases but clearly there are large variations as the figures show. The reason for the sudden drops is that at a certain point increasing the budget leads to the replacement of many CPs by only one, because the latter traces more variables as compared to all the former combined, thus leading to a decrease in the added overhead.

**Summary:** SCI collects traces from 16 benchmarks versus only six for previous work. The hardware implementation highly outperforms the software one, and gains an average coverage of 41.5% for three benchmarks that the software implementation fails to extract any data from. The hardware implementation decreases the WCET overhead compared to software to 21.0%, and decreases code size overhead to 31.9%.

## VIII. Discussion

This section focuses on some high-level issues concerning the applicability of the techniques described in this work.

**Usefulness of Partial Instrumentation:** Time-aware instrumentation constraints the instrumentation process and extracts only partial information. A full instrumentation, however, can still be constructed from the union of multiple partial instrumentation instances. This might not be convenient in some cases, but will allow timely execution of the partially instrumented versions of the software versus a full instrumentation. Apart from constructing a full instrumentation, the extracted partial information is still useful [20], [21].

**Instrumentation of Multiple WCPs:** Our analysis ignores the rare case of multiple WCPs existing in a program. This case did not appear in any of our experiments. However, addressing multiple WCPs is an easy task. We mentioned earlier that if the WCP changes after instrumentation, then the tool will convert the IPs on the new WCP to CPs. The tool can simply extend this concept to directly instrument multiple WCPs with CPs.

**Concurrent Applications:** For concurrent applications, the static slack $\alpha$ is an allowable increase in the WCET of each task such that the schedulability relations hold. While the specific way to distribute static slack among the tasks is up to the developer, the whole workload must remain schedulable.

**Choosing a WCET Analysis Tool:** Our analysis uses RapiTime [18] to obtain the WCET of basic blocks. RapiTime is a measurement-based WCET analysis tool and thus might underestimate the actual WCET. WCET, however, is only an input to our framework and thus the validity of the proposed concept is independent of the accuracy of the analysis tool. The choice of RapiTime was due to the availability of the tool in our labs, past experience using it, and independence of the architecture on which the software executes.

**Rerunning the WCET Analysis:** The number of instrumentation retries is usually low [13]. We mentioned in Section II that our tool discovers WCP changes through rerunning the WCET analysis. This is also required to ensure that the new WCET, after instrumentation, is still below the deadline. If the new WCET exceeds the deadline, the tool will decrease the static time window $\alpha$ and rerun the instrumentation process.

**Limitations of SCI:** In this work, we focused on tracing scalar variables. It is possible to extract other information such as array elements, function calls, or branches.

This work assumes that the overhead of a CP is less than the WCET of the instrumentation code in the CP. Otherwise, replacing the CP with an IP would have less overhead. Usually, the instrumentation code involves reading variables from memory and either writing these variables to memory buffers or sending them off-chip. Hence, CPs usually have less overhead than their instrumentation code. Otherwise, our tool can be modified to replace a CP with an IP in that case.

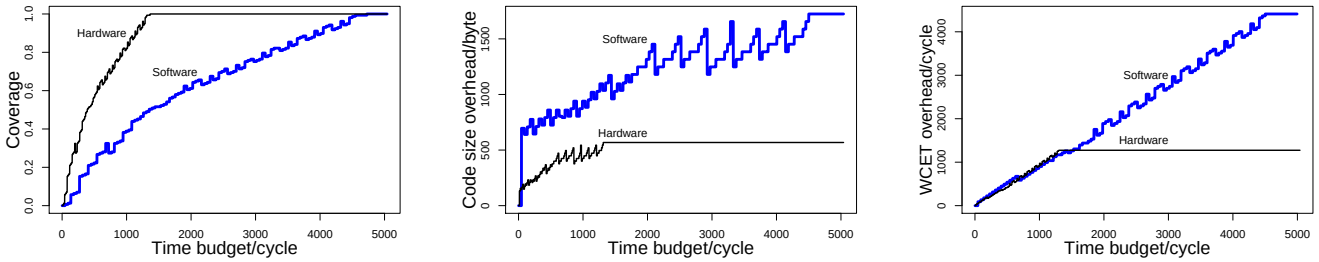
In some cases, the minimization of CPs decreased the instrumentation coverage (Figure 8a). This conflicts with the fact that delaying instrumentation and reducing CPs accumulates more slack to increase coverage. This requires slightly varying the budget to achieve the best instrumentation coverage.

## IX. Related Work

*Static instrumentation* frameworks include Atom [3], Etch, Morph, and Executable Editing Library (EEL). *Dynamic instrumentation* tools include DynamoRIO [6] and Pin [5]. None of these tools support concepts of interference or constraints such as timing and resource bounds.

*Hardware-based methods* include using performance counters, special monitoring hardware [8], [9], or simple emulation such as in-circuit emulation (ICE) hardware. Although such approaches naturally provide low interference, they can still have a significant impact on performance [10].

*Static time-aware instrumentation* techniques instrument a program at code locations that do not modify the program's WCET (or modify it within a given budget). The authors




(a) Instrumentation coverage variation (b) Increase in code size variation (c) Overhead on the WCP variation

Fig. 8: C-MINCP for the matrix inversion algorithm

in [12] apply code transformation techniques to increase instrumentation coverage. The authors in [13] introduce INSTEP; an instrumentation framework for preserving multiple competing extra-functional properties. None of these techniques, however, can instrument the WCP of a program. In this work, we build on top of [11], and, as a future extension, we can complement our work with the techniques in [12] and [13].

DIME [7], a framework for *time-aware dynamic binary instrumentation*, uses rate-based resource allocation to limit the instrumentation time to a pre-specified budget. DIME is well-suited for instrumenting soft real-time applications. We focus on static source-code instrumentation techniques that are better suited for instrumenting hard real-time applications.

Kritikakou et al. [22] propose a run-time WCET controller for ensuring the predictability of concurrently executing high criticality tasks. The proposed run-time control mechanism, used to monitor a task's execution time, is more sophisticated than ours in in handling loops and function calls. However, the goal of our work is different, and the specific approach to run-time control is orthogonal to the concept of SCI.

## X. Conclusion

In this work, we investigate an SCI mechanism that obeys timing constraints but can also extract information on the WCP. While previous work exists on tracing, these approaches only preserve logical correctness or preserve timing as well but cannot extract information on the WCP. We compared hardware and software-based implementations in detail and proposed solutions to two problems of how to efficiently use SCI. We also reported on non-intuitive results such as the negative side effects of minimizing the number of instrumentation points. Overall, however, our approach improves over previous work on time-aware instrumentation by an order of magnitude in instrumentation coverage and several orders of magnitude in the number of executed instrumentation points (=generated trace data) at the expense of code size.

## References


[1] A. Ko and B. Myers, "A Framework and Methodology for Studying the Causes of Software Errors in Programming Systems," *Journal of Visual Languages & Computing*, vol. 16, no. 1-2, Apr. 2005.

[2] M. Gallaher and B. Kropp, "The Economic Impacts of Inadequate Infrastructure for Software Testing," National Institute of Standards & Technology Planning Report 02–03, May 2002.

[3] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *SIGPLAN Not.*, vol. 39, April 2004.

[4] H. Thane, "Monitoring, Testing and Debugging of Distributed Real-Time Systems," Ph.D. dissertation, Department of Computer Science and Electronics, Mälardalens University, May 2000.

[5] C.-K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, 2005.

[6] D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Washington, DC, USA, 2003.

[7] P. Arafa, H. Kashif, and S. Fischmeister, "Dime: Time-aware dynamic binary instrumentation using rate-based resource allocation," in *Proc. of the 13th International Conference on Embedded Software (EMSOFT)*, Montreal, Canada, Sep 2013.

[8] L. J. Moore and A. R. Moya, "Non-Intrusive Debug Technique for Embedded Programming," in *Proc. of the 14th International Symposium on Software Reliability Engineering (ISSRE)*. Washington, DC, USA: IEEE Computer Society, 2003.

[9] W. Omre, "Debug and Trace for Multicore SoCs," ARM, Tech. Rep., Sep. 2008.

[10] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. Sweeney, "We have it Easy, but do we have it Right?" *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008.

[11] S. Fischmeister and P. Lam, "Time-Aware Instrumentation of Embedded Software," *IEEE Transactions on Industrial Informatics*, Aug. 2010.

[12] H. Kashif and S. Fischmeister, "Program transformation for time-aware instrumentation," in *Proc. of the 17th IEEE International Conference on Emerging Technologies & Factory Automation (ETFA)*, Sep 2012.

[13] H. Kashif, P. Arafa, and S. Fischmeister, "INSTEP: A Static Instrumentation Framework for Preserving Extra-functional Properties," in *Proc. of the 19th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Aug 2013.

[14] R. Wilhelm et al., "The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, 2008.

[15] M. Velasco, P. Martí, J. M. Fuertes, C. Lozoya, and S. A. Brandt, "Experimental evaluation of slack management in real-time control systems: Coordinated vs. self-triggered approach," *J. Syst. Archit.*, vol. 56, no. 1, Jan. 2010.

[16] P. Lokuciejewski, D. Cordes, H. Falk, and P. Marwedel, "A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models," in *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009.

[17] D. August et al., "UNISIM: An Open Simulation Environment and Library for Complex Architecture Design and Collaborative Development," *IEEE Comput. Archit. Lett.*, vol. 6, July 2007.

[18] RapiTime. http://www.rapitasystems.com/products/RapiTime.

[19] SNU Benchmarks. http://www.cprover.org/goto-cc/examples/snu.html.

[20] M. Serrano and X. Zhuang, "Building Approximate Calling Context from Partial Call Traces," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO, Washington, DC, USA, 2009.

[21] G. Pothier, E. Tanter, and J. Piquer, "Scalable Omniscient Debugging," in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007.

[22] A. Kritikakou, C. Rochange, M. Faugère, C. Pagetti, M. Roy, S. Girbal, and D. G. Pérez, "Distributed run-time wcet controller for concurrent critical tasks in mixed-critical systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14. New York, NY, USA: ACM, 2014.