INSTEP: A Static Instrumentation Framework for Preserving Extra-functional Properties

Hany Kashif, Pansy Arafa and Sebastian Fischmeister
Dept. of Electrical and Computer Engineering
University of Waterloo, Canada
{hkashif, parafa, sfischme}@uwaterloo.ca

***Abstract*—Tracing is a well-established method for debugging programs. Current approaches aim only at preserving functional correctness during the instrumentation. Preservation of functional correctness is a necessary feature of all instrumentation tools. However, few existing instrumentation tools preserve extra-functional properties of a program. Specific classes of software are unable to leverage software instrumentation; e.g., timing for real-time systems, memory consumption for embedded software, and tracing bandwidth for on-board software.**

We present the first instrumentation framework, INSTEP, that preserves logical correctness and a rich set of extra-functional properties. INSTEP derives instrumentation alternatives based on the developer's instrumentation intent (*II*), abstracts the program and prunes the search space, and then instruments the program based on constraints and cost models of competing properties. We demonstrate and experiment with a fully automated framework of INSTEP with different *II*s and extra-functional properties. We also experiment with a large automotive case study to show the scalability of INSTEP.

I. Introduction

Tracing is a well-established method for analyzing and debugging programs. Developers use this technique to diagnose the faulty behavior of software programs. Program instrumentation is the modification of program code to trace its execution and extract information at run time. A necessary feature of the instrumentation process is maintaining the logical correctness of the program after instrumentation. However, preserving the functional correctness *alone* is often insufficient.

Software systems are rich in extra-functional (or non-functional [1], [2]) requirements such as timing, code sizes, communication bandwidth, power consumption, and memory consumption. Current instrumentation techniques are unfit for such systems, because these techniques ignore such extra-functional properties. Consequently, using a current instrumentation framework for such systems can introduce side effects that produce unintended behavior. For instance, embedded software run on microcontrollers that might have limited on-chip RAM. Instrumenting such software programs might lead to exceeding the memory limit. Another example is time-sensitive programs in the field of real-time embedded systems. Instrumentation of a real-time program might cause it to exceed its time budget or deadline.

Changing the location of the instrumentation code in a program can have an effect on the extra-functional properties. Consider, for instance, the function in Listing 1. The function prints the value of z in the if and else statements, and prints w before returning. Calling this function 10 million times has an execution time of around 2.29 seconds on a 2.5GHz dual core platform. The `printf()` calls can be slightly modified by removing the calls at labels *A* and *B*, and printing both z and w in the call at label *C*. This reduces the execution time to around 1.75 seconds. Hence, the proper placement of instrumentation code can affect the performance and thus can help meet extra-functional properties like timing. Similar examples for other properties like binary size or communication channel throughput are straightforward.

```
    int simple(int x, int y){
        int z,w;
        if (x % 2 = 0){
            z = x + 100;
A:          printf("%d\n",z);
        } else {
            z = x * 9;
B:          printf("%d\n",z);
        }
        w = (z / 2) * y;
C:      printf("%d\n",w);
        return w + z;
    }
```

Listing 1: Simple Example

Maintaining an extra-functional property during instrumentation is complicated and managing multiple properties simultaneously is even more so. Extra-functional properties can be competing where meeting one property might break another. The instrumentation framework needs to weigh and trade off such competing properties. As an example, assume that the instrumentation framework can choose from several variables to instrument and locations in the source code where to instrument them. It might be the case that instrumenting, for example, six variables at one location minimizes the instrumentation time. However, at the same time, the system might have insufficient bandwidth to store and communicate the six variables at once and thus splitting up the instrumentation would be favorable. At that point minimizing both execution time and memory bandwidth is impossible.

This work presents a static instrumentation framework that gives the developer unprecedented control over what to instrument and what to preserve. It thereby presents the first fully-implemented instrumentation mechanism that considers multiple competing extra-functional properties. INSTEP uses trees to represent instrumentation intents (*II*s) and automata to represent cost models. The work provides insight into pruning the search space of instrumentation alternatives to find a feasible instrumentation. The experiments demonstrate the usage

of *II*s and cost models together with four different constraints and objectives.We experimented with multiple benchmarks as well as an industrial automotive module. The experimental results show the accuracy of INSTEP in honoring constraints and demonstrate its practicality and scalability.

The framework is directly applicable to a variety of use cases, including debugging and testing. In testing and oracle selection, the number of test inputs required to achieve a certain level of fault finding can be reduced through selecting an oracle that has a higher percentage of internal program variables [3]. However, increasing the number of internal variables used by the oracle, increases perturbation to extra-functional properties which may lead to violation of some constraints. Therefore, it is essential to choose an oracle which reduces test cases but at the same time preserves constraints.

II. Extra-Functional Instrumentation Overview

INSTEP is a static instrumentation framework that considers multiple competing extra-functional properties while instrumenting a program. Developers start by specifying their instrumentation intents. *II*s specify variables of interest, their weights, and logical relations among them. Section III explains *II*s in more detail. The *II* specification allows the framework to extract at run time the information which is most valuable to the developer. INSTEP also permits developers to specify cost models for the different extra-functional properties of interest. A cost model is a weighted automaton that assigns different costs to actions like variable instrumentation and variable bit-width assignment. This helps INSTEP to maximize or minimize certain properties and satisfy constraints on others.

Figure 1 shows the block diagram for INSTEP. The framework operates in two phases: (1) the partial program derivation phase and (2) the determinising instrumentation phase. In the partial program derivation phase, INSTEP transforms the input program into a partial program based on the *II*s. A partial program is one containing non-deterministic choices which have to be resolved [4].

The partial programs in INSTEP contain possible alternatives of where to execute the instrumentation. In the determinising instrumentation phase, INSTEP transforms the non-deterministic partial program into a deterministic instrumented program. This transformation is based on cost models for competing extra-functional properties together with any constraints on any property. In this phase, the framework attempts to find a feasible solution that satisfies all constraints and maximizes the objectives (or minimizes them based on their semantics).

Separating the framework into two phases fosters modularity and reuse. The use of partial programs as an intermediate representation shows promise [4] and allows for the modularity of the design to support extensions like different language processors or different back-ends. One possible extension, for example, is generating multiple instrumented programs that cover all the *II*s (in case one is not enough) and at the same time honor constraints. Another reason is re-using the partial program for instrumentation after, for example, relaxation of cost models or constraints.

The development of the framework involved solving a set of challenges that are specific to its two phases of operation. Section III discusses the partial program derivation phase. It specifies the instrumentation intent representation and addresses the first challenge of deriving instrumentation alternatives from the *II*s to create a partial program. Section IV discusses the determinising instrumentation phase. It describes the representation of automata-based cost models and gives examples of various extra-functional properties. It also addresses the second and third challenges. The second challenge is pruning the search space of the partial program. The third challenge is the formulation of an optimization problem from the cost models, the constraints, and the partial program. Solving this optimization problem yields a deterministically instrumented program.

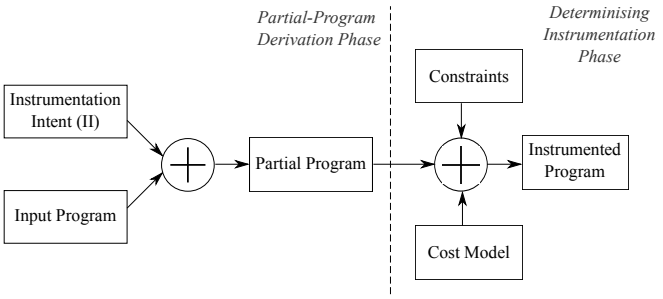


Fig. 1: Extra-functional instrumentation framework

III. Partial Program Derivation

In this phase, INSTEP uses two inputs: (1) the input program and (2) the instrumentation intents (*II*s). With these, INSTEP extracts the program's CFG and generates instrumentation alternatives based on the *II*s. This section describes the inputs and the generation process in detail.

A. The Input Program

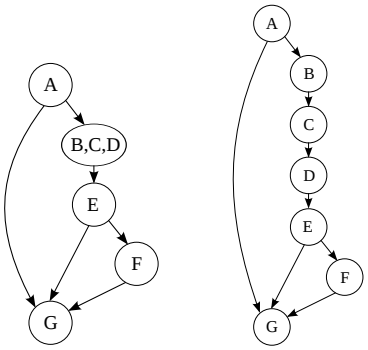
INSTEP uses CIL [5] to extract the input program's control-flow graph (CFG). INSTEP supports data structures in MISRA C [8] compliant programs (more details in Section VI). INSTEP supports advanced constructs such as nested statements and recursive functions. A CFG is a directed graph $G = \langle V, E \rangle$ which captures the program's inter-procedural control flow. Each vertex $v \in V$ in the program's CFG represents a basic block in the program. A basic block is a unit of execution in the program and has a single entry and exit point. In our model, each basic block contains at most one assignment to any variable of interest. Thus, if a basic block in the original CFG contains two assignments for variables mentioned in the *II*s, then our model will split it into two basic blocks. Each block will contain only one of the assignments and one block will follow the other. The set of edges, $E \subset V \times V$, represent the flow of control in the CFG.

Listing 2 shows a sample input program which is part of the *sqrt* benchmark from the SNU benchmark suite [6]. Figure 2a shows the CFG for the input program before splitting any basic blocks. Basic block $\langle B, C, D \rangle$ contains all three statements *B*, *C*, and *D*. Assuming that the *II*s contain the variables *dx, x, val, diff, flag*, Figure 2b shows the modified CFG after splitting basic block $\langle B, C, D \rangle$ into three separate

basic blocks B, C, and D. INSTEP uses this modified CFG for its transformations.

```
A: if(!flag){
B:     dx = (val−(x*x))/(2.0*x);
C:     x = x + dx;
D:     diff = val−(x*x);
E:     if(fabs(diff) <= min_tol){
F:         flag = 1;
       }
   }
G:
```

Listing 2: Input Program



(a) CFG (b) Modified CFG

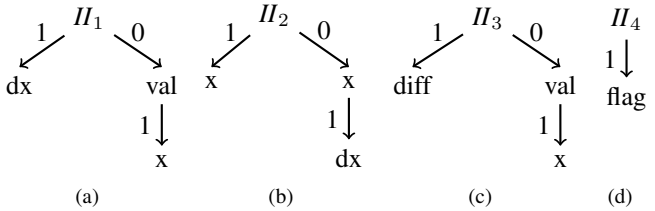
Fig. 2: Input program CFGs

B. The Instrumentation Intent

The input *II*s in INSTEP represent a set of required instrumentations specified by the developer. An *II* follows a tree structure specifying variables of interest specific to this particular *II* and values representing the importance of these variables. The tree of an *II* specifies a logical relation between the variables. For example, consider statement B in Listing 2. If a developer wants to trace variable *dx*, he might be interested in either variable *dx* or variables *val* and *x*. This intent has the following propositional logic expression: $(dx \vee (val \wedge x))$. As Figure 3a shows, the *II* for this statement consists of two branches where the ANDed variables lie on the same branch, and the ORed variables lie on different branches. The developer assigns the values based on the importance or usefulness of the variables. The particular *II*s in Figure 3a only uses values 1, 0, and 1 for the variables *dx*, *val*, and *x*, respectively. This encodes that variable *val* alone is useless without variable *x* and that variables *val* and *x* have an equal value to *dx*. Figure 3 shows the *II*s for the statements B, C, D, and F of the input program in Listing 2.

A node in an *II* tree can contain more information than just a variable's name. For example, in Figure 3b INSTEP requires a separation in the *II* between variable *x* on the left hand side (LHS) of statement C and variable *x* on the right hand side (RHS). Line numbers are also required to identify the locations of the variables.

***II* Specification.** An *II* can originate from different sources. During a debugging session, the developer will most likely specify the *II*. A simple tool based on program slicing could generate *II*s from a high-level specification. This engineering effort, however, is outside the scope of this work. If variables are equally important, the developer can leave the *II* values at their defaults as in Figure 3a (experimentation in this paper uses default values). Testing tools and tracing tools can also generate the *II*s based on a high-level specification [7]. The results show that our framework is robust and tolerates inaccuracies in the model.



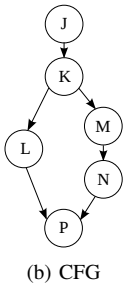
(a) (b) (c) (d)

Fig. 3: Instrumentation intent

C. The Derivation

After extracting the input program's CFG and parsing the *II*s, INSTEP finds instrumentation alternatives for the different variables in the *II*s. An instrumentation *alternative* is one or more locations in the code where a variable can be instrumented to extract its desired state before it changes. Normally, an instrumentation alternative is a single basic block (one location) at which a variable can be instrumented (recall that each basic block contains at most one assignment to any variable of interest). An alternative, however, can be multiple basic blocks (more than one location). Consider the example in Figure 4 which shows a code snippet and its corresponding CFG. The variables of interest are *x*, *w*, and *z*. One instrumentation alternative of variable *x* is at the end of block *J* (one location). Another alternative, can be after the if-condition (block *K*) and before the branch sink *P*. Blocks *L* and *M* (together) can be instrumented to cover all sub-paths between nodes *K* and *P* and provide a valid instrumentation alternative. Therefore, if an alternative is comprised of multiple basic blocks, this means that all these blocks have to be instrumented to represent a valid instrumentation of the variable.

```
J: x = 2;
K: if(y > 3){
L:     z = x*4;
   } else {
M:     w = x*3;
N:     z = w + 4;
   }
P:
```



(a) Code (b) CFG

Fig. 4: Branch example

The instrumentation engine finds locations in the modified CFG that permit instrumentation. The engine coarsely follows the following rules:

1) **Variable is on the LHS of a statement:** INSTEP inserts

instrumentation alternatives in the current block and all following blocks until the variable's value is overwritten.

2) **Variable is on the RHS of a statement:** INSTEP inserts instrumentation alternatives from the last variable's assignment prior the current basic block until the next block that overwrites the variable's value.
3) **Variable is on both sides of the statement:** (as in statement C of Listing 2) INSTEP inserts instrumentation alternatives from the last variable's assignment until the current use of the variable in the statement (alternatives for the RHS). It also inserts alternatives from the current assignment of the variable until the variable's value gets re-assigned (alternatives for the LHS).

Note that these are only the coarse rules used by the engine and that the implementation contains more detailed rules. The exact locations of the alternatives depend on the type of the basic blocks, branches in the CFG, etc. Consider a variable *var* that is assigned in a basic block *B*. To fulfill the aforementioned rules, the instrumentation engine should be able to traverse the CFG upwards (towards the root) and downwards (away from the root) from *B* to find instrumentation alternatives for *var*. Algorithm 1 briefly describes how the instrumentation engine finds alternatives moving downwards from *B*. The algorithm for traversing the CFG upwards is omitted for conciseness and space constraints. We describe the operation of Algorithm 1 along with a few cases out of around 366 different cases that INSTEP covers.

Algorithm 1 takes as input: the CFG G, the variable *var* for which instrumentation alternatives are to be found, and the basic block B in which *var* exists. The CFG can have blocks of the following types: branch (if or switch statement) which has more than one child, loop-start, loop-break, return (a return statement), and instruction (instructions with no branches). The function *addAlt* adds the start or end of a basic block as an instrumentation alternative. Recall that an alternative can be multiple basic blocks. Algorithm 1 may also find alternatives for one of the multiple blocks that form an alternative. For instance, consider the example in Figure 4. If Algorithm 1 is finding alternatives for variable *x*, then after storing blocks *L* and *M* as one alternative, it will store *N* as an alternative for *M*. Hence, when storing an alternative, function *addAlt* keeps track of which block the alternative is for (details are omitted from the algorithm for clarity). The *enqueue* operation used in the algorithm, will only enqueue a block if it was not marked as visited. The algorithm starts by instrumenting the end of the input block B and enqueueing its child (an instruction block always has one child). The dequeued block is handled according to its type.

We describe a few cases that explain the operation of Algorithm 1. Consider, for example, Listing 2 and its CFG in Figure 2b. If INSTEP is finding instrumentation alternatives for variable *diff*, then the first instrumentation alternative will be the end of basic block D. Normally, children of an if-block are an alternative as well, however, in this case, the if-block E has a branch sink G which is also one of its children. Instrumenting the child F alone is not an alternative, because it will leave subpath $\langle E, G\rangle$ without instrumentation. Hence, INSTEP bypasses the if-block and chooses its sink G as an alternative. Note that this will only be possible, if no subpath modifies the variable *diff*. Another example is: if INSTEP finds the start of a loop, then INSTEP will continue finding alternatives following the break of the loop only if the variable is not modified inside the loop. A third example is: if the algorithm encounters a loop-break (without first encountering a loop-start), this means that block B is inside a loop. The algorithm, in that case, will not find alternatives beyond the loop break because information is missed by instrumenting outside the loop.

Algorithm 1 Find Alternatives Downwards

Input: CFG, instrumentation variable *var*, basic block B
Output: Instrumentation alternatives

```
1: Let Q be an empty queue
2:
3: Call addAlt(end of B) and mark B as visited
4: Enqueue B's child in Q
5: while Q is not empty do
6:    Dequeue B from Q
7:    if B is a loop-start then
8:       Let C be the loop-break block
9:       Enqueue C's child in Q if the loop does not modify var
10:   else if B is an instruction block and does not modify var then
11:      Call addAlt(end of B)
12:      Enqueue B's child in Q unless a back edge connects them
13:   else if B is a branch source then
14:      Let C be the branch sink (if exists)
15:      If var is not modified between B and C, enqueue C in Q
16:      If C is not B's child, mark C as visited
17:      if C does not exist or C is not B's child then
18:         for Each block D in B's children do
19:            if D is an instruction block then
20:               if D does not modify var then
21:                  Call addAlt(end of D) and enqueue D in Q
22:               else
23:                  Call addAlt(start of D)
24:               end if
25:            else
26:               Call addAlt(start of D) and enqueue D in Q
27:            end if
28:         end for
29:      end if
30:   end if
31:   Add B to the set of visited vertices
32: end while
33: return Instrumentation alternatives
```

D. The Partial Program

The partial program is an intermediate representation that contains all possible instrumentation alternatives. The derivation phase of the framework inserts the instrumentation alternatives in the input program to generate the partial program. Listing 3 shows the partial program for the input program in Listing 2 after INSTEP inserted the instrumentation alternatives. Note that the notation in the listing is only for illustration purposes, because an instrumentation alternative must hold more information. For example, the framework needs to know which alternatives must simultaneously exist if on parallel branches. For the statement C, Listing 3 uses $x.l$ and $x.r$ to differentiate between the left and right x variables, respectively. (II_1, val), for example, represents a location where the variable val from II_1 can be instrumented.

IV. Determinising the Instrumentation

In this phase, INSTEP processes three inputs: (1) the partial program from the first phase, (2) constraints on the instrumentation, and (3) cost models for instrumentation methods. INSTEP uses these three inputs to formulate an optimization problem and attempts to solve it using local searching [9] to find a feasible solution. A feasible solution is a selection of the instrumentation alternatives that satisfies the constraints, and maximizes or minimizes other objectives that may exist. This section describes the constraints, the cost models, the formulation of the optimization problem, and the final output of INSTEP.

```
   (II1,val),(II1,x),(II2,x.r),(II3,val)
A: if(!flag) {
      (II1,val),(II1,x),(II2,x.r),(II3,val)
B:    dx = (val-(x*x))/(2.0*x);
      (II1,dx),(II1,val),(II1,x),(II2,x.r),(II2,dx),(
           II3,val)
C:    x = x + dx;
      (II1,dx),(II1,val),(II2,x.l),(II2,dx),(II3,val)
           ,(II3,x)
D:    diff = val-(x*x);
      (II1,dx),(II1,val),(II2,x.l),(II2,dx),(II3,diff
           ),(II3,val),(II3,x)
E:    if (fabs(diff) <= min_tol) {
F:       flag = 1;
         (II4,flag)
      }
   }
G: (II1,dx),(II1,val),(II2,x.l),(II2,dx),(II3,diff),(
     II3,val),(II3,x),(II4,flag)
```

Listing 3: Partial Program

A. Specifying Constraints

Constraints are restrictions on the instrumented program that might prevent it from achieving the maximum value of the instrumentation intent or the highest output for any other objective. Constraints primarily pose limits on some extra-functional properties. For example, one constraint may be a limit on the code size of the program after instrumentation. Another constraint might be a limit on the memory consumption while running an instrumented program. Finally, an upper limit of the debugging budget added to the worst-case execution time (WCET) for the instrumentation [10], [11] is another form of constraint. Enforcing such constraints on the instrumentation process requires knowledge of the cost functions. A cost model specifies the cost for the different aspects of instrumentation. Modern modeling systems like UML/MARTE and AADL facilitate specifying the different constraints and cost models by the developer.

B. Cost Models

In this context, cost models are simply weighted automata that describe costs of actions. The development of the cost models themselves is out of the scope of this work. Figure 5 shows two cost models for instrumentation points. Figure 5a represents a code size cost model for adding a printf() instrumentation point for Integers on an ARM Cortex-M3. The cost for the first variable is 14 bytes of code and 8 bytes for each extra variable that can be added in the same printf() statement. The first variable has a cost of 14 because it includes

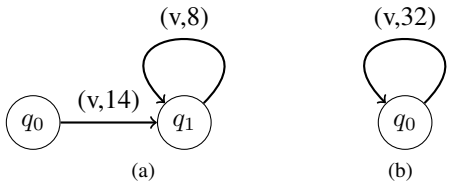


Fig. 5: Cost models

instructions for a function call which are added once for an instrumentation point of this type. Figure 5b shows a code size cost model for writing the instrumented variable to a buffer array using *gcc* on an Intel Core i5-2520M CPU. Each variable would add 32 bytes of code to the program.

The values in a cost model depend on many factors. Example factors include the used hardware, the type of instrumentation (e.g., printf() to the serial port of a chip, saving to a buffer, sending over TCP/IP), and the type of variable (e.g., integer, double, character). There might also be an overhead cost for including a library required for instrumentation (to use the network stack for example). The use of automata provides a general concept that, for instance, also supports extending INSTEP to consider caches and costs of reading from cache versus main memory [4]. This work considers cost models for code size of instrumentation points, timing of instrumentation points, and detection latency. A cost model for the timing of a printf() instrumentation point may look like that of code size. Detection latency is the latency between assigning the variable and instrumenting it.

Accuracy of Cost Models. Cost models are widely used to estimate performance costs of certain operations such as writing to memory and data transmission [4], [12]. It is clear that adding cost models for all details of the target architecture complicates the analysis but provides more precise results. We assess the effect of using inaccurate cost models on the output of INSTEP. This is discussed in more detail in Section V.

C. The Formulation

INSTEP combines the cost models, the constraints, and the partial program into an optimization problem. In its current form, INSTEP supports four main extra-functional properties: *II* values, code size, execution time, and detection latency. Each of these can be used in objectives and constraints. For example, a developer can choose to minimize code size or set an upper limit as a constraint. It is easy to extend INSTEP to consider types of variables, different instrumentation types, and other properties such as memory consumption and tracing bandwidth (for TCP/IP). The formulation in the examples below uses printf() to a serial port as the instrumentation type.

1) Decision Variables: The framework creates a boolean variable for each instrumentation variable in the *II*s indicating whether the variable is instrumented or not. It also creates a boolean variable for each instrumentation alternative to indicate whether the alternative is chosen or not. Variable *flag* in Listing 3, for example, has two instrumentation alternatives so the framework will create three variables; *II4_flag*, *II4_flag_F_e*, and *II4_flag_G_s* denoting the instrumentation of variable *flag*, instrumenting *flag* at the end of node F, and

instrumenting *flag* at the start of node G, respectively. For each variable, the framework creates the following constraint:

$$-var + \sum alternatives \geq 0$$

This encodes that the variable is instrumented only if one of its alternatives is chosen. For variable *flag* in Listing 3 the constraint will be:

$$-II4\_flag + II4\_flag\_F\_e + II4\_flag\_G\_s \geq 0$$

2) II *Values Property:* INSTEP will maximize the total value of all *II*s or meet a minimum value, if the developer specifies a constraint. The value of an *II* is the maximum of the values of all variables in an *II* tree. The value of a variable is the summation of the values on the path leading to the variable's node in the *II* tree from its root. This value of a variable is only realizable if it is instrumented and all its ancestor variables as well. If a variable exists more than once in a tree, the maximum of these values is taken. For example, considering II_1 in Figure 3a, its value is equal to:

$$V1 = max(II1\_dx * 1, II1\_val * 0, II1\_val * II1\_x * (0 + 1))$$

This shows that instrumenting variable *val* alone is useless and also reflects that variable *x* is useful only if *val* is instrumented as well.

3) Code Size Property: As for the timing and code size, cost models specify the costs of instrumentation for these two extra-functional properties. Hence, each instrumentation point that INSTEP inserts in the code will have a cost that needs to be considered. For example, consider the instrumentation point at the start of node A in Listing 3. It has four variables available for instrumentation. To formulate the cost in terms of code size, consider the cost model in Figure 5a. Simply, if any of the variables is instrumented at that point, a cost of 14 bytes will be incurred and 8 bytes for each extra variable. This can be formulated as follows, taking into account that a variable can appear more than once at the same instrumentation point in a partial program:

$$\begin{aligned} code\_A\_s = (14 - 8) * (&II1\_val || II1\_x || II2\_x.r || II3\_val) \\ &+ 8 * (II1\_val || II3\_val) + 8 * (II1\_x || II2\_x.r) \end{aligned}$$

This computes the extra overhead for the first variable of the instrumentation point if any variable is instrumented. It also incurs the cost of a variable only once if it exists in multiple *II*s. INSTEP formulates a similar cost function for each instrumentation point with regards to timing.

The framework represents the total code size after instrumentation as the total of (1) original code size, (2) any overhead for using printf() instrumentation (cost incurred if any instrumentation point exists), and (3) the cost of all instrumentation points with respect to code size. The developer can specify minimizing code size as an objective or set a limit on the total code size that should not be exceeded.

4) Execution Time Property: As for timing, to respect a given debugging time budget, INSTEP requires knowledge of the WCET of the program, the WCET of the different basic blocks, function calls, and the worst-case number of iterations of each basic block. This work assumes the presence of correct but maybe conservative WCET analysis tools, which is a common assumption for most works in real-time systems. INSTEP extracts function calls through static analysis, and obtains all other information through the RTBx data logger and RapiTime [13], which is a measurement-based WCET analysis tool. The timing of the main function of the program from start to end, after instrumentation, can be either minimized as an objective or be constrained with a developer-specified debugging budget. To formulate the effect of instrumentation on the timing of the code, INSTEP formalizes a cost of a function as the maximum timing of all paths in the function. This formulation is a conservative approximation as it ignores cache effects, branch prediction, etc. Taking the maximum of all paths requires enumerating all paths which is exponential. Therefore, INSTEP traverses the CFG of a function and instead of enumerating paths, it takes the maximum of subpaths from a branching source node to its sink. This is a practical over-approximation and worked well in the experiments.

INSTEP also prunes paths according to the following rules:

- If the subpaths between a branch source and sink, do not have instrumentation points or function calls, INSTEP will prune the max function to the subpath with the largest timing.
- If a subset of the subpaths has instrumentation points and/or function calls, INSTEP will only consider this subset along with the largest timing subpath.

INSTEP can further prune the CFGs (through abstracting them for example), but this will only be effective for complex timing cost models that include more architecture-related information.

The cost of a function is equal to the cost of its basic blocks multiplied by the number of iterations of the blocks and taking the maximum of subpaths in case of branches. The cost of a block is equal to the WCET of the block, the WCET of any instrumentation point in the block (from the cost model), and any function calls in the block. The cost of a function call is simply equal to the cost of the called function. Note that the cost of calling and returning from a function is already part of the WCET of the calling block. The cost of the function in Listing 2 and Figure 2b would be:

$$\begin{aligned} func = &(A + t\_A\_s) * W_A * I_A + (B + t\_B\_s + t\_B\_e) \\ &* W_B * I_B + (C + t\_C\_e) * W_C * I_C + (D + t\_D\_e) \\ &* W_D * I_D + E * W_E * I_E + (F + t\_F\_e) * W_F * I_F \\ &+ (G + t\_G\_s) * W_G * I_G \end{aligned}$$

where t_B_s and t_B_e, for example, are the costs of the instrumentation points at the start and end of node B, respectively. In practice, the path $\langle E, G \rangle$, for example, might be worse, with respect to timing, than the path $\langle E, F, G \rangle$; however, this is an approximation that INSTEP uses in its current form. If the WCET of the instrumented program exceeds the specified debugging budget, INSTEP will find the instrumentation point causing the violation (increase in WCET of block does not match expected increase from cost model), remove the instrumentation point and rerun the analysis to ensure that constraints are met. The experiments show that the violations do not often occur, and if they do, the number of retries is low. This is because INSTEP can use one WCET

analysis report (provides WCET of basic blocks) to detect multiple violations of the instrumentation process.

5) Detection Latency Property: The detection latency of a variable is the minimum of the detection latencies of its instrumentation alternatives. A function like $1/x$ can represent this property where x is the amount of time from the specified variable location until its instrumentation. Since the instrumentation is at the granularity of the basic blocks, the detection function will only be defined at possible instrumentation locations (start and end of blocks). A developer's goal might be minimizing latency, i.e., maximizing the detection function or specifying a constraint on the detection latency. Several paths might exist between a statement and the instrumentation alternative. In such case, the maximum latency of the different subpaths is chosen. If a variable has ancestors in the *II*, then the detection latency is the maximum of its detection latency and latencies of all its ancestors. The detection latency of an *II* is the minimum across all its nodes.

D. The Instrumented Program

After INSTEP formalizes the optimization problem, it uses local search to find a feasible solution. Local search is used because the problem is highly non-linear with a large number of decision variables resulting from the instrumentation alternatives. Such combinatorial model is out of the scope of current state-of-the-art solvers relying on classical tree-search techniques [9]. Local search attempts to find candidate solutions by applying local changes in the search space. The solution would be an assignment to the defined boolean variables which can be easily used to transform the partial program into an instrumented program. Local search either returns an infeasible solution, if the constraints can never be satisfied, a feasible solution, or an optimal solution. A solution is infeasible if, for example, the specified constraint for code size is below the original code size. If the input constraints are at least equal to the corresponding values of the input program, i.e., if, for example, WCET constraint is at least equal to the input program WCET, then local search will always find a feasible solution.

For our experiments, we used the standard setup for local search and it worked reasonably well as Section V demonstrates. The pseudo-random number generator seed is set to zero. The simulated annealing level is set to one. The search is parallelized over two threads. The experimental results show the applicability and practicality of our approach. Finding the best configuration parameters for local search is out of the scope of this work.

V. Experimentation

This section presents experimentation using the fully automated framework INSTEP.

A. Experimental Setup

There are three sets of experiments:

1) We experiment with the SNU real-time benchmark suite [6]. It contains 17 C benchmarks that have 120 lines of code on average, and implement numeric and DSP algorithms.
2) We run an experiment on the web server example [14] for NXP LPC17xx ARM-based microcontrollers. This program implements a dynamic web server and has a total of 1,846 lines of C code.
3) We conduct an experiment on an automotive control module. It has 177,298 lines of C code and 6,297 basic blocks. This number excludes definitions in header files, since the industrial partner provided only parts of the overall application. Consequently, the experiments on this program only show the scalability of INSTEP and applicability to industrial code, without showing the results on the WCET analysis.

The benchmarks are run on a Keil MCB1700 board running a 100 MHz ARM Cortex-M3 processor-based MCU.

Metrics: We quantitatively evaluate the accuracy and precision of the framework using the following metrics:

- **WCET of the instrumented program:** We run a WCET analysis for the instrumented program. The WCET should be less than or equal to the input program's WCET plus a specified debugging time budget.
- **Number of retries:** If the WCET of the instrumented program exceeds the debugging time budget, INSTEP will remove the instrumentation points that cause a higher WCET than expected. Finding these instrumentation points is straightforward. The WCET analysis tool outputs the WCET of each basic block before and after instrumentation, and basic blocks that now violate the constraints are immediately visible. INSTEP then reanalyzes the WCET of the modified program. We report the number of retries required to produce the final instrumentation.
- **INSTEP execution time:** The size of the inputs (e.g., number of *II*s, code size) can increase the time that INSTEP needs to generate the instrumented program. This metric indicates the applicability of INSTEP to large-scale software programs used in industry.
- **Number of instrumentation alternatives in the partial program:** As the code size of the input program increases, it becomes more challenging to derive an instrumented program honoring the extra-functional properties while maximizing objectives. A large program offers multiple locations for instrumenting a variable which also complicates the optimization problem.

Extra-functional Properties: The experimentation considers four extra-functional properties: the *II* values, code size, execution time, and detection latency. We use a printf() to the serial port of the microcontroller for instrumentation. Thus, the code size cost model is the one shown in Figure 5a. The cost for the first variable is 14 bytes of code and 8 bytes for each extra variable that can be added in the same printf() statement. Additionally, there is an overhead cost of 460 bytes for including the library required for instrumentation. A cost model for the timing of a printf() instrumentation point is similar to that of code size but with different values. The first variable in a printf() statement costs 4,000 cycles, and each extra variable in the same printf() statement costs 3,850 cycles as measured on the target platform. Finally, the function $1/x$ represents the cost model for detection latency. Execution

time and code size properties are considered constraints to the optimization problem. We limit the debugging budget (constraint) by a 10% increase in the WCET of the input program [15]. The code size constraint is arbitrarily chosen to be an additional 554 bytes to the input program size. 554 bytes are the size of the input library plus five separate printf statements where each instruments a single variable. The optimization has two objectives: maximizing *II* values and minimizing the detection latency.

Choice of* IIs:* Each SNU benchmark is run with 1,100 different inputs and has two different sets of *II*s: (1) a maximum of 30 input *II*s, and (2) a maximum of four *II*s. The *II*s were randomly chosen to avoid any bias in the experimental results. Note that only six out of 15 benchmarks had enough variables to form 30 *II*s. For the rest of the benchmarks and for the first set of the *II*s, the maximum number of *II*s were input to INSTEP. The web server experiment has 79 *II*s as input. This is the maximum number of *II*s available in the web server software. The automotive module experiment has three versions. The objective of the first version is to instrument all assignments of an arbitrary local variable in a function, resulting in nine *II*s. The second version instruments the five most occurring global variables in the program, which is equivalent to 54 *II*s. Whereas the third version instruments the five most occurring local variables across all functions represented in 21 *II*s.

B. Experimental Results

The results of the different sets of experiments show that instrumented benchmarks do not violate any of the constraints. The results also show that INSTEP satisfies more *II*s compared to a naive instrumentation. They also demonstrate the scalability and applicability of INSTEP to industrial software.

Figure 6a shows the ratio of the WCET of the instrumented benchmark to that of the input. Figure 6b shows the increase in code size of the instrumented benchmarks. Benchmarks *select* and *sqrt* have a WCET ratio of 1 and no increase in code size for *II* Set 2. This means that INSTEP did not instrument any variables so as not to violate any of the constraints. The figures show that the increase in both the WCET and code size are within the specified constraints. Note that benchmarks bs and $insertsort$ are omitted from the results, because inserting any instrumentation point in any of them would violate a constraint. Hence, INSTEP left these two benchmarks intact without instrumentation.

INSTEP vs Naive Instrumentation*:* The code size constraint was set such that it comprises five printf() statements with a single variable each. This means that a naive instrumentation will most probably satisfy five *II*s at most. Out of 34 experiments, 17 had more than five *II*s. In 14 out of these 17 experiments, INSTEP satisfied more than five *II*s with a maximum of 26 for the automotive module. This indicates the strength of the framework in finding alternatives and merging them to satisfy the most *II*s and honor constraints. Figure 7 also shows the percentages of the input *II*s that INSTEP was able to satisfy. For some benchmarks, the satisfied number of *II*s in the second set is less than four, while being much larger for the first set. This depends on which subset of *II*s from the first set are chosen for the second. It might be the case that the chosen subset of *II*s violate constraints thus satisfying none of the *II*s as it is the case with *select* and *sqrt* benchmarks.

Retries*:* Retries were required in 16 out of 38 experiments. In most of the cases, one retry was required and at most four retries were needed with an average of 0.85 retries. The number of retries is low due to the ability of INSTEP to find (and remove) multiple instrumentation points causing a violation to the debugging budget using one WCET analysis report. The reason is that one WCET analysis report is sufficient to detect violations in each basic block of the program's CFG. The low number of retries shows the practical feasibility and viability of INSTEP even for large programs.

Execution Time*:* The execution time increases as the input program has a more complicated or larger CFG, more *II*s, etc. The execution time has a reasonable average of 2.64 seconds with a maximum of 32 seconds for instrumenting the automotive module. The reported time does not include the time required for applying local search. Figure 6c shows the time required by local search to find the best reported feasible solution within 10 minutes. In 30 out of 38 experiments, local search found the solution in only one second. All other times in the figure appeared only once with a maximum of 261 seconds. This shows that a satisfactory solution can be found in a reasonable amount time.

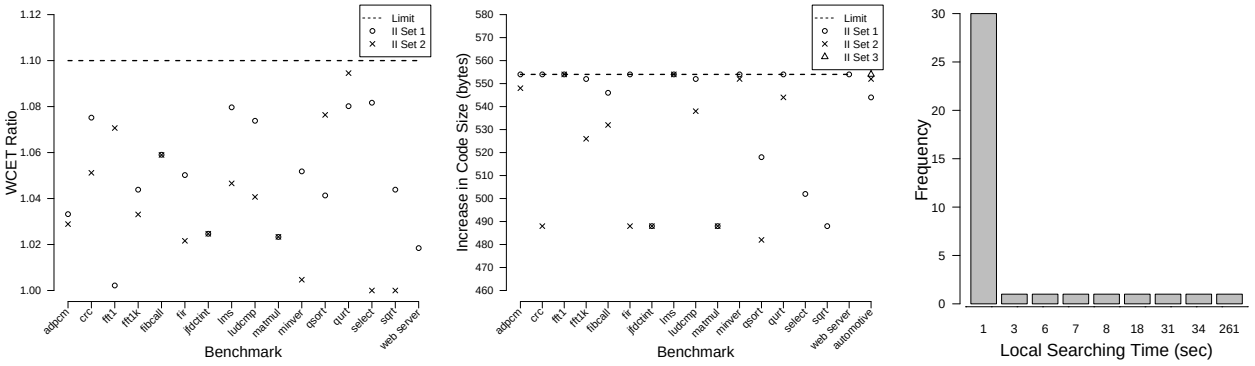
Instrumentation Alternatives*:* The number of equations and expressions in the optimization problem exceeded 1,500 and 33,000, respectively which shows the complexity of the problem. The number of alternatives reached 3,041 which indicates the efficiency of the tool in finding multiple alternatives. It also shows that INSTEP scales and can accommodate this large number of alternatives within reasonable time limits. INSTEP also handles the large number of equations and expressions, and local search finds satisfactory solutions that satisfy a large number of *II*s in an acceptable time frame.

Inaccuracy of Cost Models*:* To test the effect of inaccurate cost models on the output of INSTEP, the experiments were repeated using modified cost models for time and code size. Two modified versions of the cost models were used: (1) underestimated models which reduce the cost of transitions in the original time and code size models by 500 cycles and 2 bytes, respectively, and (2) overestimated models which increase the cost of transitions by the same values for both of the original models. Underestimating the cost model can lead to more violations which increase the number of retries. In 60% of the experiments, the number of retries did not change, and increased by only one retry in the rest of the cases. Overestimating the cost model might reduce the number of satisfied *II*s. In 80% of the experiments, the number of satisfied *II*s did not change, and decreased by only one otherwise. This shows that inaccuracies in the cost models can be tolerated by INSTEP which can still output satisfactory results.

VI. Discussion

This section discusses some issues regarding the limitations and applicability of this work.

Logical Correctness. INSTEP preserves the logical (functional) correctness of a program after instrumentation. The



(a) WCET ratio (b) Increase in code size (c) Local search

Fig. 6: WCET ratio, code size increase, and local search time of instrumented benchmarks

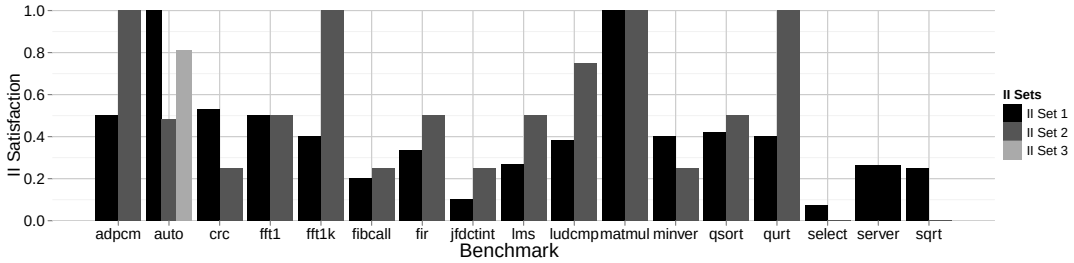


Fig. 7: *II* satisfaction

only modification that INSTEP makes to a program is the insertion of instrumentation points. These instrumentation points only read variables from memory locations and there is no concurrent variable access. For all experiments, the outputs from the instrumented programs matched those of the uninstrumented programs.

MISRA C Compliance. INSTEP supports instrumenting data structures in MISRA C [8] compliant programs which restricts, for instance, the usage of pointers and unions. MISRA C also restricts the usage of dynamic memory allocation (`malloc()`). Extending INSTEP to consider memory consumption of the software as an extra-functional property is therefore restricted to static memory allocation.

Concurrency. With the current WCET analysis tools in place, INSTEP only supports instrumenting foreground/background systems and multi-programming systems with run-to-completion semantics. Concurrency complicates computing the budgets and testing whether a block will exceed its budget; however, the underlying concepts still apply and can be extended given the available tools.

Hardware Tracing. One limitation on hardware tracing is that some systems do not support hardware debugging. Hardware tracing also offers traces at a low system level, e.g., instruction level. This makes software tracing more suited to debugging at a higher object level, e.g., debugging a task control block. Another aspect is the evolution of the tracing mechanism along with the software being debugged. If the software is modified or updated, a software tracing mechanism can be easily maintained along with it, as opposed to a hardware tracing module or device.

Partial Tracing Information. In this work, the examples and experiments focus on tracing data variables. Similarly, INSTEP can trace control flow and function calls. INSTEP focuses on extracting information while preserving the input program's extra-functional properties. This definitely limits the amount of information that INSTEP can extract from the program. INSTEP, however, attempts to maximize the satisfied *II*s as the experiments demonstrate. Note also that partial traces are useful for analyzing and understanding programs, as well as for optimizations [16]. Moreover, INSTEP is easily extensible to generate multiple instrumentations of the same input program to satisfy all *II*s (if possible). This allows extracting more tracing information but from different program runs (which is a limitation to the debugging process).

VII. Related Work

Manual instrumentation is highly flexible, the developer [19], however, can not estimate its underlying effect on the extra-functional properties of the program. An automatic instrumentation tool parses the program, may generate a CFG, and inserts instrumentation points. GCC profiling and code coverage tool is an example of an automatic instrumentation tool. Binary-level instrumentation tools insert instrumentation points into binary executables either statically or dynamically during the program execution. Static binary instrumentation tools include Morph [20] and EEL [21]. Dynamic instrumentation tools rewrite the instructions executed at run time based on the instrumentation specification. Pin [22] and DynamoRIO [23] are examples of dynamic instrumentation tools which do code transformation during program execution.

Other software and hardware instrumentation methods [24], [25], [26] were proposed to enable tracing and system logging. All the instrumentation approaches mentioned so far are poor at maintaining the extra-functional properties of the program. They affect the program's behavior especially the temporal behavior [11]. Partial instrumentation can be used to respect timing constraints [16]. Fischmeister et al. [10] presented time-aware instrumentation which honors the original program's timing, especially the worst case behavior. Kashif and Fischmeister [11] apply program transformation techniques to increase the effectiveness of time-aware instrumentation.

Multi-Objective Compilation. Naik and Palsberg [17] present a framework for code-size-aware compilation. They formulate register allocation as an Integer Linear Programming (ILP) problem. In our optimization problem, the model of the input *II*s and the cost models introduce non-linearity. Moreover, merging instrumentation alternatives to reduce overhead of instrumentation points makes the problem highly non-linear. Therefore, the authors' approach is inapplicable in our case.

Lee et al. [18] introduce a framework to balance the trade-offs between code size, execution time, and energy consumption when developing an embedded system. The framework satisfies the design constraints and assigns code/WCET pairs to the tasks to minimize the system cost function. In our problem, the alternatives for instrumenting variables are huge reaching more than 3,000 alternatives for one of our test cases. In addition to choosing the alternatives that satisfy the most *II*s, INSTEP also picks the least number of variables that satisfy these *II*s. This makes our problem harder because the choice of an alternative might lead to a smaller code size, for instance, if it satisfies more than one *II*. In the work by Lee et al., however, the effect of choosing a code/WCET pair is clearly tangible on the overall code size and the schedulability of the system. This is different from our problem.

VIII. Conclusion

Current tracing and instrumentation tools only preserve functional correctness of the program. Unfortunately, some application domains require tools that not only consider the functional correctness, but also consider extra-functional properties such as timing. In this work, we propose INSTEP; an instrumentation framework that preserves extra-functional properties. To generate the instrumented program, INSTEP derives a partial program based on the developer's *II*. Then, it formulates an optimization problem according to the input cost models and constraints, and solves the problem using local search. The design of INSTEP allows for the reusability of partial programs and for future extensions. INSTEP, in its current state, honors four extra-functional properties.We conducted experiments on benchmarks as well as an industrial automotive module. The experimental results show the accuracy and precision of the tool in honoring constraints. They also show the practicality and scalability of INSTEP.

References

[1] J. Sincero, W. Schroder-Preikschat, and O. Spinczyk, "Approaching Non-functional Properties of Software Product Lines: Learning from Products," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, 2010.

[2] D. Lohmann, W. Schroder-Preikschat, and O. Spinczyk, "Functional and Non-functional Properties in a Family of Embedded Operating Systems," in *The Tenth IEEE International Workshop On Object-Oriented Real-Time Dependable Systems (WORDS 2005)*. IEEE Computer Society.

[3] M. Staats, M. Whalen, and M. Heimdahl, "Better testing through oracle selection: (NIER track)," in *Software Engineering (ICSE), 2011 33rd International Conference on*, may 2011.

[4] P. Černý, K. Chatterjee, T. A. Henzinger, A. Radhakrishna, and R. Singh, "Quantitative synthesis for concurrent programs," in *Proceedings of the 23rd international conference on Computer aided verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011.

[5] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *In International Conference on Compiler Construction*, 2002.

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

[7] S. Navabpour, B. Bonakdarpour, and S. Fischmeister, "Optimal instrumentation of data-flow in concurrent data structures," in *Proceedings of the 15th international conference on Principles of Distributed Systems*, ser. OPODIS'11. Berlin, Heidelberg: Springer-Verlag, 2011.

[8] MIRA Ltd, *MISRA-C:2004 Guidelines for the use of the C language in Critical Systems*, MIRA Std., Oct. 2004. [Online]. Available: www.misra.org.uk

[9] T. Benoist, B. Estellon, F. Gardi, R. Megel, and K. Nouioua, "Localsolver 1.x: a black-box local-search solver for 0-1 programming," *4OR: A Quarterly Journal of Operations Research*, vol. 9, 2011.

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

[11] H. Kashif and S. Fischmeister, "Program transformation for time-aware instrumentation," in *Proc. of the 17th IEEE Intl. Conf. on Emerging Technologies & Factory Automation (ETFA)*, Krakow, Poland, 2012.

[12] X. Tang, J. Wang, K. B. Theobald, and G. R. Gao, "Thread partitioning and scheduling based on cost model," in *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, ser. SPAA '97. New York, NY, USA: ACM, 1997.

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

[14] EasyWeb. http://www.keil.com/download/docs/295.asp.

[15] M. B. Dwyer, M. Diep, and S. Elbaum, "Reducing the cost of path property monitoring through sampling," in *Proceedings of the 2008 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008.

[16] M. Serrano and X. Zhuang, "Building Approximate Calling Context from Partial Call Traces," in *Proc. of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '09. Washington, DC, USA: IEEE Computer Society, 2009.

[17] M. Naik and J. Palsberg, "Compiling with code-size constraints," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 1, Feb. 2004.

[18] S. Lee, I. Shin, W. Kim, I. Lee, and S. L. Min, "A design framework for real-time embedded systems with code size and energy constraints," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, Jan. 2008.

[19] B. Simon, D. Bouvier, T.-Y. Chen, G. Lewandowski, R. McCartney, and K. Sanders, "Common Sense Computing (Episode 4): Debugging," *Computer Science Education*, vol. 18, no. 2, 2008.

[20] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System Support for Automatic Profiling and Optimization," *SIGOPS Oper. Syst. Rev.*, vol. 31, 1997.

[21] J. R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," *SIGPLAN Not.*, vol. 30, 1995.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proc. of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, 2005.

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

[24] M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky, "JavaMaC: A Run-Time Assurance Approach for Java Programs," *Form. Methods Syst. Des.*, vol. 24, no. 2, 2004.

[25] 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.

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