Time-aware Instrumentation of Embedded Software

Sebastian Fischmeister, *Member, IEEE,* and Patrick Lam

***Abstract*—Software instrumentation is a key technique in many stages of the development process. It is particularly important for debugging embedded systems. Instrumented programs produce data traces which enable the developer to locate the origins of misbehaviours in the system under test. However, producing data traces incurs runtime overhead in the form of additional computation resources for capturing and copying the data. The instrumentation may therefore interfere with the system's timing and perturb its behavior.**

In this work, we propose an instrumentation technique for applications with temporal constraints, specifically targetting background/foreground or cyclic executive systems. Our framework permits reasoning about space and time and enables the composition of software instrumentations. In particular, we propose a definition for trace reliability, which enables us to instrument real-time applications which aggressively push their time budgets. Using the framework, we present a method with low perturbation by optimizing the number of insertion points and trace buffer size with respect to code size and time budgets. Finally, we apply the theory to two concrete case studies: we instrument the OpenEC firmware for the keyboard controller of the One Laptop Per Child project, as well as an implementation of a flash filesystem.

***Index Terms*—Instrumentation, tracing, debugging, real-time systems.**

I. Introduction

Instrumentation and tracing are key activities in debugging microcontroller-based embedded systems. Instrumented programs produce data traces, which developers can use to diagnose misbehaviors in the system under test. For example, if a trace shows incorrect control flow at a conditional branch, then the branching condition or the input values influencing that branch will be most likely causes for the bug.

However, instrumentation and tracing incur runtime overhead. The consequences of the instrumentation overhead range from negligible to devastating: while some systems tolerate changes in code timing, heavily-loaded real-time applications often do not tolerate such changes. Our approach aims at instrumenting real-time programs by considering time budgets and imposing minimal overhead.

Related work on instrumentation has not considered time budgets. Current software instrumentation frameworks—built for monitoring executions of programs for non-embedded systems—typically insert code immediately after each occurrence of a traceable event. For instance, the AspectJ [1] and Etch [2] instrumentation and monitoring frameworks enable developers to monitor every write to a heap variable, but do not have any provision for monitoring subject to constraints on overhead. We compare more instrumentation frameworks in Section XI.

This work concentrates on time-aware instrumentation of Misra-C compliant functions in a background/foreground [3] or cyclic executive system [4], [5]. Misra-C provides a standard for implementing safety-critical real-time systems; for example, it requires bounded loops and limits recursion.

Technically, background/foreground systems or a cyclic executive with interrupts are preemptive multi-tasking systems with exactly two tasks. The background task always executes and consists of a single endless loop, sometimes called a *super loop*, which invokes a collection of functions in sequence. For example, in a keyboard controller, these functions steps can be (a) handle command, (b) update LEDs, (c) check power savings mode, (d) process received messages, and (e) update watchdog timer. The foreground task preempts the background task whenever a serviceable interrupt line becomes asserted. The background task never preempts the foreground task. For example as part of the foreground part, an interrupt can handle urgent actions such as emptying the receive buffer whenever a new message has been received for the keyboard controller. In this work we assume no nested interrupts and one interrupt priority level, yet our results still hold for the target class of systems: generalizing to nested interrupts and interrupt priority levels requires refining calculating the execution time c and consider blocking times due to nested, prioritized interrupts.

The key idea behind the *time-aware instrumentation* of a system is to transform the execution-time distribution of the system so as to maximize the reliability of the trace while always staying within the time budget. Our notion of reliability implies that the instrumentation will provide useful data over longer periods of tracing. A time-aware instrumentation injects code, potentially extending the execution time on all paths, while ensuring that no path takes longer than the specified time budget.

The time budget is the worst-case execution time of a function without violating a specification. In hard real-time systems, the time budget can be the longest execution time without missing any deadline, and depending on the longest execution time of the non-instrumented version, more or less time will be available for the instrumentation. In systems without deadlines, the time budget can be the current maximum execution time plus a specified non-zero maximum overhead for tracing to the current maximum execution time.

Figure 1 shows the expected consequences of time-aware instrumentation in a hard real-time application on the probability density function of a loop iteration's execution time. The x-axis specifies the execution time of the loop iteration, while the y-axis indicates the frequency of the particular execution time. The original uninstrumented code has some arbitrary density function. We have chosen the Gaussian distribution for this example for illustrative purposes; Li et al. provide details from empirical observations of distribution functions [6]. The distribution for the instrumented version differs from the original one. It is shifted towards the right, but still never passes the deadline. This shift occurs because time-aware instrumentation adds to paths, increasing their running times, but ensures that execution times never exceed the deadline.

Note that our execution-time model concentrates on the overhead involved in acquiring data. A related problem is to transport the collected data from the embedded system to an external analysis unit. While that problem admits many solutions, one common solution is to piggyback the buffer information onto serial or network communication.

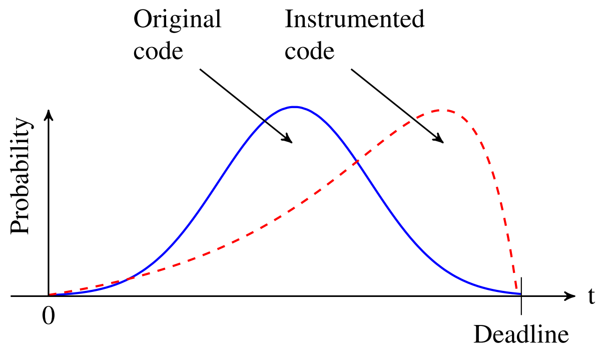


Fig. 1. Execution-time distribution for a code block before and after time-aware instrumentation showing the shift in the expected execution time.

So, what do we need to perform time-aware instrumentation? First, we need an underlying model which captures relevant properties from the source code. Since we concentrate on timed systems, this model should include the temporal behaviour of the system as well as its system's control flow. It should also indicate which data needs to be logged. The model then allows us to calculate the impact and effectiveness of various instrumentations. For example, we can use the model to calculate how the execution time will change on each control-flow path. Our goal, however, is to use the model to determine the optimal instrumentation for runtime traces. Optimal means that, given a time budget, the system provides the best instrumentation possible in terms of instrumentation reliability.

The contributions of this paper include:

- a definition for instrumentation reliability;
- a definition of "time-aware instrumentation", which instruments code, optimizing for code space and reliability, while meeting specified time bounds;
- strategies for computing time-aware instrumentations with and without temporal bounds;
- an implementation of a research framework that enables experiments on the impact of time-aware instrumentation; and
- experimental results exploring the impact of time-aware instrumentation on the OpenEC keyboard controller code and an embedded flash filesystem implementation.

II. Methodology

We propose the following instrumentation stages:

- **Source analysis:** The source-code analyzer breaks the functions into basic blocks and generates a call graph. The analyzer also presents a list of variables which are assigned in these basic blocks and the developer can choose a subset of these variables to trace. For hard real-time applications, the analyzer annotates the call graph using execution time information obtained through static analysis or measurements [7].
- **Naive instrumentation:** Using the control-flow graph, the execution times of the basic blocks, and the input variables for the trace, we inject code into the selected function at all instrumentation points.
- **Enforce time budget:** If the naive instrumentation exceeds the time budget, we use the technique in Section VI-E to compute an instrumentation which does respect the time budget while maximizing the reliability of the instrumentation.
- **Minimize code size:** If the instrumentation is reliable enough, then we apply semantics-preserving, decreasing transformations (Section VII) to reduce the size of the instrumented code.
- **Collect traces:** The developer finally recompiles and executes the instrumented program.

Figure 2 shows the workflow that results from the steps. To instrument a function, we start by picking the function of interest. We then use the assembly analyzer to extract the control flow graph and break the function into execution paths. In the first phase, we use a tool to instrument all variables of interest and then check whether the execution time on the worst-case path has changed. If it has changed, then we will use integer linear programming to lower the reliability of the instrumentation so that it meets the timing requirements. If the reliability is too low, then we can either give up, if we cannot extend the time budget available for the function and the instrumentation; or extend the time budget, which will allow for higher-reliability instrumentations. If the optimized instrumentation meets the required reliability, or if the initial naive instrumentation does not extend the worst-case path, then we will proceed and use the identified execution paths to minimize the required code size. Afterwards, we can recompile the program and collect the desired traces from the instrumentation.

III. Motivating Example

We illustrate the contributions of this work by applying them to the OpenEC source code. OpenEC [8] is an effort to implement an open firmware for the embedded controller of the XO platform (from the One Laptop per Child project). OpenEC is currently development-stage code; as of October 2008, the source consists of 8090 lines of C code with inline assembler.

The OpenEC code conforms to the background/foreground structure. Listing 1 shows the main loop of the OpenEC source. In this loop, the program sequentially calls the main function blocks. At the end of the loop, the controller will suspend itself for the amount of time remaining in its budget. The typical loop frequency is 100Hz. Thus, the time budget for the main loop is 10ms. The function sleep_if_allowed suspends the keyboard controller until 10ms have elapsed since the start of loop.

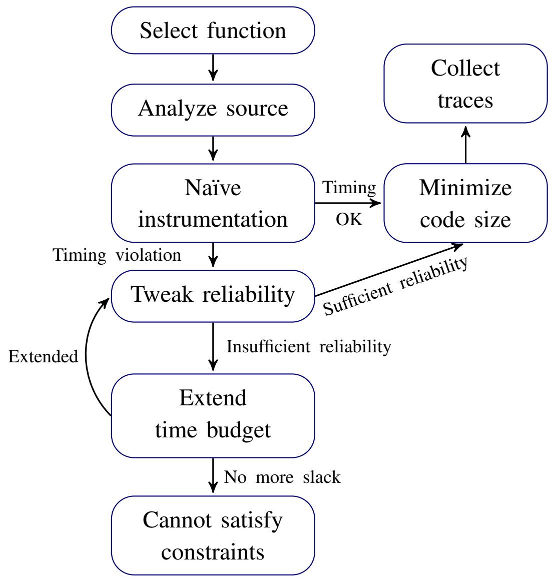


Fig. 2. Workflow of applying time-aware instrumentation.

```
while(1)
{
  STATES_TIMESTAMP();

  busy = handle_command();
  busy |= handle_cursors();
  handle_leds();
  handle_power();
  handle_ds2756_requests();
  handle_ds2756_readout();
  busy |= handle_battery_charging_table();

  watchdog_all_up_and_well |= WATCHDOG_MAIN_LOOP_IS_FINE;

  print_states();
  monitor();
  handle_debug();
  sleep_if_allowed();
}
```

Listing 1. Main loop of the OpenEC source.

The One-Wire bus and the debugging UART generate incoming interrupts for the foreground tasks. We bound their effect by considering the bit rates of the bus and UART.

A subtask of the background task handles the power button. This subtask switches the main XO machine on and off as appropriate (and also handles, for instance, various LEDs and the wireless networking subsystem). Listing 2 presents part of the code for handling the power button. We will demonstrate the instrumentation process for this procedure.

```
void handle_power(void) {
    if(power_private.my_tick == (unsigned char)tick)
        return;
    power_private.my_tick = (unsigned char)tick;

    switch(power_private.state) {
        case 0:
            if( POWER_BUTTON_PRESSED ) {
                power_private.timer++;
                if( power_private.timer == HZ/10 ) {
                    LED_PWR_ON();
                    power_private.state = 1; } }
            else power_private.timer = 0;
            break;
        case 1:
            SWITCH_WLAN_ON();
            power_private.state = 2;
            break;
    /* ... */
    STATES_UPDATE(power, power_private.state);
```

Listing 2. Source excerpt: button handler.

Listing 3 presents case 1 of the switch statement in 8051 assembler. We propose the instrumentation of the assembler code; it suffices to add instrumentation code after the instruction `movx r0, a`. To carry out time-aware instrumentation, we compute the cost of the procedure and instrument it if the budget allows. If the budget does not allow for complete instrumentation, we instrument the subset of the writes to memory which maximizes reliability and optimizes for code size, and then report on the reliability of our instrumentation.

```
;       power.c:219: case 1:
00112$:
;       power.c:220: SWITCH_WLAN_ON();
        mov     dptr,#_GPIOD00
        movx    a,@dptr
        mov     r2,a
        orl     a,#0x02
        movx    @dptr,a
;       power.c:221: power_private.state = 2;
        mov     r0,#(_power_private + 0x0002)
        mov     a,#0x02
        movx    @r0,a
        ; ** instrument power_private.state here **
;       power.c:222: break;
        ljmp    00172$
```

Listing 3. Compiled power button code.

IV. Model Definition

We abstract the source program as a directed graph $G = \langle V, E \rangle$, representing the program's interprocedural control flow, and use functions $c : V \to \mathbb{R}$ and $p : E \to [0, 1]$ to model the program's behaviour. For background/foreground implementations and cyclic executives, G contains a large cycle ("super loop"), representing the forever-running task, with subtasks on the spine of the large cycle.

An *instrumentation* of a software program inserts custom code at specific insertion points into a program. An *instrumentation operation* is the piece of code that realizes the desired instrumentation function at the insertion point. A *uniform instrumentation* inserts the same instrumentation operation at each insertion point. A *complete instrumentation* inserts code at every insertion point, while a *partial instrumentation* only inserts code at some insertion points. Finally, an instrumentation is *stateless* if it decides whether to instrument an insertion point deterministically and solely based on the code immediately before that insertion point.

In our use case, we always instrument the program for assignment tracing. That is, our instrumentation operation copies a variable's value into a buffer. The contents of the buffer are then read after the program terminates or at the bottom of the super loop.

A. Static Analysis Approach

We have built a static analysis tool which accepts C programs and extracts relevant data, including the control-flow graph, basic blocks, and a cost model. Figure 3 presents the structure of our static analysis tool; we next summarize the structure and discuss our key design decisions.

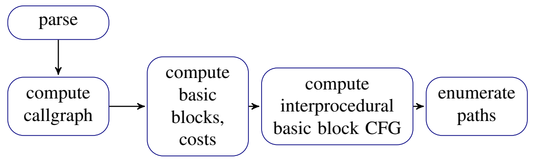

parse
compute callgraph
compute basic blocks, costs
compute interprocedural basic block CFG
enumerate paths

Fig. 3. The source analysis step preceding the instrumentation step in the workflow.

We analyze assembler code directly. Our case study was written for an 8051-family microcontroller, which is simple to model: it suffices to count the (constant) number of cycles each instruction takes to execute. We decided that the benefits of having an exact cost model and the ease of parsing assembler outweighed the benefits of getting structured programs.

Our tool parses the assembler code emitted by SDCC [9], the C compiler for the OpenEC project. Next, it computes the call graph and basic blocks. Since our target class of embedded programs is free of function pointers and dynamic dispatch, we were able to use a straightforward callgraph construction algorithm. Finally, it computes an interprocedural control-flow graph for the program, based on the call graph and individual control-flow graphs for each procedure. Our interprocedural analysis is context-insensitive: it matches procedure `p`'s return statement with all callers to `p`.

Our abstraction enables us to enumerate the set of paths between two program points and to compute the cost of each path. Recall that we assume a general loop in the task which executes forever (therefore no timer or interrupt-driven periodic task implementation); our approach allows us to enumerate the paths between the beginning and the end of the super loop, as well as between other arbitrary program points. Our approach can also simulate the effect of finite loop unrolling while enumerating paths, by allowing a bounded number of visits to the loop decision points. In general, microcontroller systems' loops execute a fixed number of times: loops are most often used to copy data between (fixed-size) array buffers during input and output.

B. Abstraction Definition and Timing

Each vertex in G represents a basic block in the program. We abstract a vertex $v \in V$ by $\langle A, L\rangle$, with assignments A and variables L logged at v. The function $c : V \to \mathbb{R}$ specifies the required computation time c, or cost, for vertex v. For example, $c(v_0) = 12.2t$ means that the basic block at vertex v_0 requires 12.2 time units for its execution. The cost of a path $c(p)$ is the execution time of a particular path which is the sum of costs of all vertices in the path. Edges $e := \langle v_s, v_d\rangle$ specify transitions from source vertex v_s to destination vertex v_d. The function $P : E \to [0, 1]$ gives the probability $P(e)$ that the execution will use edge e to leave vertex v_s. So, $P(\langle v_0, v_1\rangle) = 0.5$ means that on average every other execution will continue at vertex v_1 after executing v_0.

In general, it is the developer's responsibility to estimate the cost function c and the probability function P. We believe that it is fairly straightforward to estimate both of these functions for our target class of systems.

Our current analysis framework helps developers compute c by calculating cycle counts for each basic block, based on the microcontroller's specifications. More generally, microcontroller vendors provide cycle accurate simulators which allow the developer to measure the execution time of executed code. Small microcontrollers use simple structures, so the problems inherent in measuring the worst-case execution time are manageable, unlike with pipelined architectures or systems with caches which require sophisticated tools [7]. However, how our instrumentation affects the worst case on more sophisticated architectures remains an open problem. At the moment, a developer would have to rerun the WCET tools after instrumentation to ensure that cache replacement and changes in the memory layout had not invalidated the previous worst case estimates.

Developers can estimate P by collecting profiling data. Two types of branches require estimates for P: branches due to loops and branches due to conditionals. Because our target class of programs must use only bounded loops, the loop branch probabilities can easily be estimated as a function of the maximum loop iteration count. We also expect that the developer will have a test suite for the system; it is straightforward to obtain estimates for P at other conditional statements based on profiling data from the test suite, or from standard usage modelling techniques.

V. Accommodating Interrupts

Accounting for interrupts is critical: otherwise, our model of the instrumented system may meet the deadline, since there are no interrupts in the model, while the real system could miss the deadline due to interrupts.

The foreground part of a background/foreground system consists of interrupt service routines. The computation time required by the foreground part can be modelled as overhead over the normal execution time of the background part. We assume that interrupts occur as sporadic events with a known minimal inter-arrival time. We furthermore assume the interrupt service routine to be bounded and to always eventually terminate. We can then adjust the execution time of any measurement c to accommodate interrupts using response time analysis [10]. We assume the following recurrence relation for calculating the response time:

$$R(x) \text{ solution to } t^{(l+1)} = x + \sum_{irq} (\lceil t^l / f(irq_i) \rceil c(irq_i)) \quad (1)$$

with $c(irq_i)$ as the execution time of the interrupt and $f(irq_i)$ as its minimal inter-arrival rate frequency.

We can iteratively solve the recurrence relation until $t^{(l+1)} = t^l$ and can use x as the starting value for t^0. At this point, we computed the maximum response time of the background part considering all interference from interrupts. We use $R(x)$ as the solution of this recurrence relation.

As an example of overhead adjustment, consider the OpenEC's UART interrupt, which we use to retrieve generated traces. The interrupt service routine executes, in the worst case, 20 assembly instructions. Running at 32MHz and with five cycles per instruction, the interrupt service routine requires an execution time of about $c = 3\,125$ns. The UART interrupt arrives with a frequency of $11\,520$Hz resulting in $f = 0.000086805555556$. Thus, for a basic block with an execution time of $x = 10$ms, we start at $l = 0$ with

$$\begin{aligned}
t^0 &= x + c * \lceil \quad 0.01 \quad /f\rceil = 0.01036\\
t^1 &= x + c * \lceil \quad 0.01036 \quad /f\rceil = 0.01037296\\
t^2 &= x + c * \lceil \quad 0.01037296 \quad /f\rceil = 0.010373427\\
t^3 &= x + c * \lceil \quad 0.010373427 \quad /f\rceil = 0.010373443\\
t^4 &= x + c * \lceil \quad 0.010373443 \quad /f\rceil = 0.010373444\\
t^5 &= x + c * \lceil \quad 0.010373444 \quad /f\rceil = 0.010373444
\end{aligned}$$

thus stopping at $t^5 = t^4 = R(x) = 0.010373444$s.

VI. Instrumentation and Reliability

Using our refined timing model, we can calculate time budgets for systems with instrumentation. The instrumentation overhead is the sum of the computation time of the instrumentation operations at the insertion points. The calculation proceeds as follows: 1) extract the control flow paths with variable assignments for the specified function; then, 2) check whether the instrumentation stays within the time budget. If it does not, 3) compute the maximum-reliability instrumentation which respects the time budget and 4) optimize this instrumentation for code size.

In the first step, we create the set $\mathcal{P}$ of all paths p between the start and the end control-flow graph vertices of the selected function. A path is a sequence of vertices $p = v_i \rightarrow v_j \rightarrow \cdots \rightarrow v_k$ with $i \leq j \leq k$ on G.

Some instrumentation properties are impossible to monitor while respecting the system's given time budgets. The two possible solutions are either to increase the time budget or to resort to partial instrumentations:

- **Extend the time budget:** Some systems tolerate increases in their time budgets. For example, soft real-time systems [11] rely on best-effort methods to meet deadlines; no direct harm results from the occasionally missed deadline. Therefore, if some paths cause deadline misses, then we can calculate the probability that system follows deadline-missing paths, and the developer can decide whether the system tolerates this instrumentation.
- **Lower the instrumentation reliability:** Alternatively, the developer can reduce the instrumentation's reliability so that all execution paths obey the time budget. By reliability, we mean the probability that the instrumentation fails to serve the predefined purpose over a longer period of tracing. In such cases, the best we can do is to create partial instrumentations. For runtime tracing, the resulting trace may miss some variable assignments. In terms of instrumentation reliability, this means that, as we trace the system for longer periods of time, chances increase that we observe a path that lacks the instrumentation.

The concept of partial instrumentations raises the following question: What is the maximal reliability of the instrumentation for a given time budget? To explore this question, we define the notion of instrumentation reliability for partial instrumentations in the context of runtime tracing.

A. Reliability For An Insertion Point

The instrumentation reliability of an assignment x is the probability that the value assigned at x gets logged before being lost. Algorithm 1 calculates the instrumentation reliability for a runtime-tracing instrumentation insertion point v_0 using a standard depth-first search algorithm. The call $\mathrm{hit}(v_0, x, 1)$ calculates the reliability of assignment x at vertex v_0, setting pr to 1 for the initial recursive call. (pr tracks the probability of hitting node v). For a vertex v, $v.L$ denotes the set of logged (monitored) assignments at v, while $v.A$ denotes the set of all assignments (and hence all insertion points) at v. The algorithm traverses the control-flow graph until (a) it detects that a vertex logs the value of x or (b) the variable x gets reassigned. If the algorithm detects logging (case (a)), the algorithm will add the probability of the program taking that particular path to the computed reliability for v_0. If x is re-assigned before being logged (case (b)), the algorithm adds zero to the computed reliability for v_0. Otherwise, it recursively adds reliabilities for edges e out of the currently-visited vertex v by summing the outgoing reliabilities for destinations $e.v_d$, scaled by the probability pr of reaching v.

Algorithm 1 Calculate reliability of logging statement x at v.

```
procedure hit($v, x, pr$)
    if $x \in v.L$ then return $pr$ end if
    if $x \in v.A$ then return 0 end if
    $a \Leftarrow 0$
    for all $e$ such that $e.v_s = v$ do
        $a \Leftarrow a + \mathrm{hit}(e.v_d, x, P(e) \cdot pr)$
    end for
    return $a$
end procedure
```

B. Reliability For A Path

The instrumentation reliability of a path p, denoted $r(p)$, generalizes the instrumentation reliability for a single assignment by taking into account all assignments to a given set of variables on the path p. In particular, $r(p)$ is the ratio of monitored-to-missed variables along path p. Unfortunately, the intuitive approach that the reliability is simply $\frac{|\bigcup v_i.L|}{|\bigcup v_i.A|}$, using multisets, is incorrect: a vertex v can be part of many paths, so that its set $v.L$ might contain entries that are only relevant to other paths.

Algorithm 2 shows how we calculate the reliability for path p in a single pass. Essentially, as the algorithm visits a path, it stores, in l, the variables assigned but not stored in p and then adjusts one counter for misses and one for hits. The function "succ(p, n)" returns the next element after n in path p, if there exists such an element; otherwise, it returns "nil". We use variable n to iterate over the path's vertices.

Algorithm 2 Calculate the reliability of logging for a path p.

```
$n \Leftarrow \text{head}(p)$
$l \Leftarrow \emptyset$
while $n \neq \text{nil}$ do
  for all $x \in n.A$ do
    if $x \in l$ then miss $\Leftarrow$ miss $+ 1$ end if
  end for
  for all $x \in n.L$ do
    if $x \in l$ then hit $\Leftarrow$ hit $+ 1$; $l \Leftarrow l \setminus x$ end if
  end for
  $l \leftarrow l \cup n.A$
  $n \Leftarrow \text{succ}(p, n)$
end while
miss $\Leftarrow$ miss $+ |l|$
```

C. Reliability For Instrumentations

The *instrumentation reliability of a partial instrumentation*, $r(\mathcal{P})$, is the sum of the weighted reliability of all possible paths using the path probabilities as weights. The formula

$$P(p) = \prod_{e \in p} P(e)$$

gives the probability of taking a specific path $p \in \mathcal{P}$ by multiplying the probabilities for each edge $e \in p$. (In multiplying probabilities together, we assume that they are independent. Many systems, including SPIN [12], assume independent probabilities at different choice points, as we do here.)

The set $\mathcal{P}$ contains all possible paths p based on the control-flow graph G starting at the source vertex v_0. Since we define the instrumentation reliability as the expected number of logged assignments over the total number of assignments for all paths, weighted by their expected value, we get the following equation:

$$r(\mathcal{P}) = \sum_{p_i \in \mathcal{P}} r(p_i) P(p_i) \tag{2}$$

D. Timing an Instrumentation

The execution time overhead of an instrumentation depends on the software design of the capture and transfer mechanism. Our method captures values in a pre-allocated buffer (see Section VIII) and flushes the buffer at the end of the super loop. This method can incur two types of execution overhead: capturing the buffer and flushing the buffer. Note that the execution overhead varies between architectures.

- **Capturing the value:** To capture the assignment value, the instrumentation engine stores the value in a buffer.
- **Flushing the buffer:** Flushing the buffer depends on the method for moving the data off chip. Two possible methods are the JTAG tracing system or a UART interface. In both cases, the developer must compute the worst case transmission time for transmitting the buffer size.

We denote the overhead of capturing a value by o_c and of flushing the buffer with o_f. To calculate how much a path p_i contributes to the overall overhead o_p, we calculate:

$$o_p = o_f + \sum_{v \in p_i} |v.L| o_c, \tag{3}$$

and the overhead of a whole instrumentation is,

$$o_i = \sum_{p_i \in \mathcal{P}} o_{p_i} P(p_i).$$

These values constrain the reliability-maximizing optimization problem. Note that both values at this point do not consider interference as specified in Equation 1. Also o_f varies depending on the number of instrumentations along the path. On the worst-case path it will even be zero, if no variable assignments are logged. In the current work we use a single value of o_f for simplicity reasons.

E. Maximal Reliability for Constrained Time Budgets

Using the notions of reliability and time budgets, we can now address the problem of instrumenting applications which aggressively push their constrained time budgets (e.g., hard real-time applications with zero time budget). If the time budget is insufficient for a complete instrumentation, then we need to address the question: Which insertion points should we intentionally omit to maximize the information gained about the system without exceeding the time budget? Unfortunately, this problem cannot be reduced to a knapsack problem, which admits known approximation solutions, because multiple paths may share vertices, so that pruning a vertex in one path might affect the value (=path reliability $r(p)$) of another path.

We instead formulate the problem as a linear programming problem. Equation (4) shows the function to be maximized. Variables n_i store the value of the insertion point (i.e. the number of trace variables in basic block v_i). If paths share basic blocks (vertices), then the optimization function sums the coefficients of the n_is to get path costs that respect sharing.

Inequalities (5) and below represent the problem constraints: total instrumentation on each path must be less than the time budget tb after adding overhead from interrupts by calculating $R(x)$. The first two terms—$o_f + \sum_{v_i \in p_n} n_i o_c$—are a modified version of Equation (3) with n_i as $|v.L|$. Increasing the number of insertion points (i.e., setting n_i non-zero) can increase the execution time on each path. Finally, constraints (6) give boundary conditions and limit the number of variables per basic block.

$$\max \sum_{p_i \in P} \sum_{v_i \in p_i} n_i \tag{4}$$

$$R\big(o_f + \sum_{v_i \in p_0} n_i o_c + \sum_{v_i \in p_0} c(v_i)\big) \leq tb \tag{5}$$

$$\vdots$$

$$R\big(o_f + \sum_{v_i \in p_n} n_i o_c + \sum_{v_i \in p_n} c(v_i)\big) \leq tb$$

$$0 \leq n_0 \leq |v_0.A| \tag{6}$$

$$\vdots$$

$$0 \leq n_n \leq |v_n.A|$$

VII. Minimizing Insertion Points

Once we compute the maximal possible reliability for our instrumentation property, we wish to create the instrumentation which uses the minimal number of insertion points. We will use the control-flow graph G, along with the costs c and transition probabilities P, to compute such an instrumentation. Note that naive instrumentations, as seen in [13], [2], [1], do not use the minimal number of insertion points in general.

Unfortunately, finding the minimal number of insertion points is NP-hard. We can show this by reducing the NP-complete hitting set problem [14] to the instrumentation problem. Recall that the hitting set problem takes a set V of elements and a collection $\mathcal{C}$ of subsets of V; the solution is the smallest subset $H \subset V$ for which $H \cap S \neq \emptyset$ for each $S \in \mathcal{C}$. Our instrumentation problem takes a control-flow graph with a set V of vertices. Some of the vertices $x \in V$ are assignments; each such assignment has a set S of vertices where the assignment x may be logged. (This set of vertices corresponds to the nodes reachable by reassignment-free paths from x.) The instrumentation problem seeks a minimal subset H of V for which $H \cap S \neq \emptyset$ for each S. These descriptions of the problems make the reduction fairly obvious; given a hitting set instance, simply map the vertices of the hitting set instance to vertices in the control-flow graph, and map the subset S to an assignment in the instrumentation problem.

In a uniform, complete, stateless instrumentation of a non-concurrent function, an instrumentation with minimal insertion points also has a minimal increase in code size. Informally one can see this, because instrumenting a program can only increase the code size. Therefore, the program with the fewest insertion points also has the smallest code size.

Towards the minimal instrumentation. Our goal is to transform the naive instrumentation with maximal reliability so as to preserve reliability and minimize the size of the instrumentation. However, we also must ensure that the minimization does not change the time budget of the unoptimized instrumentation. We therefore propose the use of *semantics-preserving* and *decreasing* transformations on an instrumentation to minimize code size.

A semantics-preserving transformation of an instrumentation is one that keeps the same set of observable effects as the original instrumentation. An example of a semantics-preserving transformation is one that delays recording a variable, as long as the delay does not extend beyond any statement which overwrites the value to be recorded.

A decreasing transformation must not increase the number of insertion points executed in any trace of an instrumented program. The non-increasing property ensures that, if the original instrumentation did not exceed its time budget, then the transformed instrumentation also does not exceed the same time budget. An example of a decreasing transformation is one that combines two insertion points after a branch into one insertion point before the branch, as long as the branch is not a loop condition.

VIII. Minimal Trace Buffer Size

Another problem in tracing embedded programs is determining adequate sizes for trace buffers: how much data does the program need to store before the next flush at the end of the loop? The developer usually makes an ad-hoc educated guess or uses trial and error to determine whether the buffer size is sufficiently large for the given instrumentation. Our model enables developers to compute the precise size required for the trace buffer, which ensures that the trace buffer will contain all data computed during the execution.

To compute the minimal trace buffer size (which is the maximal buffer size required at run time), we extend Algorithm 2. Instead of calculating the hit and miss ratio, the modified version of the algorithm sums the storage size of the logged assignments. Specifically, instead of increasing hit by one in Line 8, we increase it by the storage size of the logged variable. If we call the modified algorithm $s(p)$, then the *maximal* buffer size is

$$b_{max} = \max(s(p_i)) \text{ for all } p_i \in \mathcal{P},$$

while the *expected* buffer size is

$$b_{exp} = \sum_{p_i \in \mathcal{P}} s(p_i)p(p_i).$$

Our prototype implementation also uses an in-buffer pointer and an identifier field to reconstruct the buffered data.

IX. Case Study I: OpenEC Case

To demonstrate the effectiveness of our approach, we have applied it to two case studies: a flash filesystem and the OpenEC controller code. In this section we describe the OpenEC case study. Specifically, we investigate the handle_power function of OpenEC. We want to trace all 20 active variables (local and global) in the function. Note that our function works similarly for tracing individual variables or flags for debugging purposes. The function handle_power consists of 42 basic blocks, with 20 different control-flow paths through these blocks. The mean execution time is 75 cycles and the worst-case execution time is 132 cycles. Monitoring a variable costs one cycle.

In the experiment, we will investigate the following two questions:

- If we tolerate zero overhead, with what reliability can we monitor variables in this function? Furthermore, what is the minimal required buffer size?
- How does the monitoring reliability change when we provide a time budget for monitoring?

To answer these questions, we implemented (a) the static analysis tool outlined in Section IV-A in Scala and (b) the ILP problem from Section VI-E in Matlab. For this case study, we assume that the time budget to be the execution time of the longest running path, $o_c = 1$, $o_f = 0$, and no interrupts.

A. Trace Reliability for OpenEC

Figure 4 presents the trace reliabilities along the different paths with the time budget equal to the longest executing path. The x-axis displays each of the individual 20 paths. The y-axis shows the monitoring reliability along each path. Since we provision for no overhead, some paths cannot be instrumented. Although the function handle_power has exactly one path using the worst-case execution time, four other paths share its control flow sufficiently that they cannot be instrumented either. Using the equations from Section VI-C and our abstraction, the monitoring reliability of the handle_power is 35.83% for this scenario.

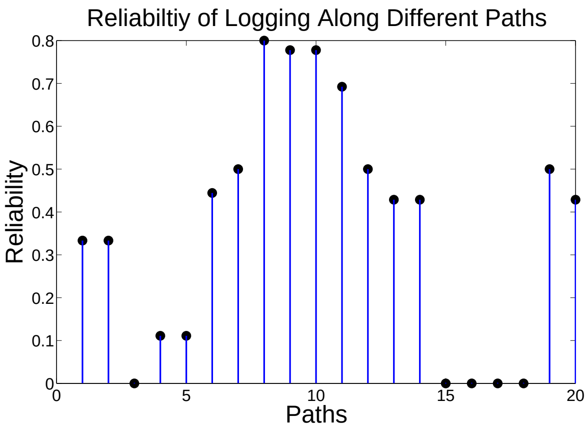


Fig. 4. Monitoring reliability of function handle_power in the 'all out' scenario.

B. Execution Time

Figure 5 shows the fitted density function for both programs. Note that this figure is only for illustrative purposes, because the execution time is a discrete function and so are all expected values. This figure illustrates what happens during instrumentation and suggests that our core idea outlined in the introduction is correct. Although the instrumentation causes only minor changes around the peak, the instrumented program still requires more execution time than the original program as one can see for example around values 120 to 140. The extent of this shift is primarily influenced by the number of assignments to heap variables outside the worst-case path. More assignments per basic block imply a more prominent shift in the density function.

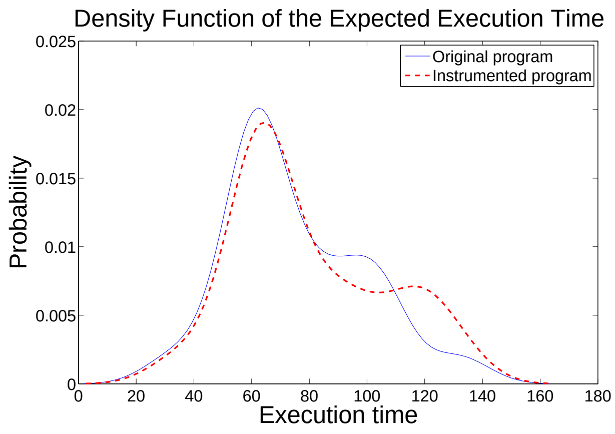


Fig. 5. Shift in the density function of the execution time of the function handle_power.

C. Minimal Buffer Size

To calculate the minimal buffer size, we use Algorithm 2 with the modifications described in Section VIII. The minimal required buffer size is independent of any particular path's tracing reliability: even if the path has a low reliability and low execution frequency, it may still eventually be executed and then the system must provide sufficient storage capacity for the trace.

The analyzed function handle_power updates only the state of the controller and sets hardware registers. All updates to the state affect variables of type unsigned char, and these variables include, for example, power_private.my_tick and power_private.timer. All updates to the registers are of the same type. Using our static analysis tool, we discover that Path 8 monitors the most variables of all paths with 16 assignments. Thus a sufficient buffer size for this instrumentation is $16 \cdot$ sizeof(unsigned char).

D. Increasing the Time Budget

In some applications, the developer can increase the time budget to devote more time to the tracing effort; consider a heavily-loaded system that drives a motor. The motor may tolerate some jitter in its duty cycle. Reliable operation, however, demands as little jitter as possible. During the monitoring effort, it might be acceptable to drive motors with functions that introduce jitter but allow for instrumentation. How much reliability can we gain by increasing our deadline by a few cycles?

The surprising result is that adding a few extra cycles to the deadline significantly increases the trace reliability. Figure 6 shows the result for the handle_power function in which we changed the deadline from 132 cycles to 137 cycles. The x-axis lists the cycles that we add to the execution time of the worst-case path. The y-axis shows the tracing reliability in the function handle_power. The reason for the surprising result is that about 25% of the paths share critical parts with the worst-case path. Thus, the algorithm cannot use the insertion points. However, relaxing the deadline provides more flexibility and the algorithm then also instruments these highly-frequented basic blocks.

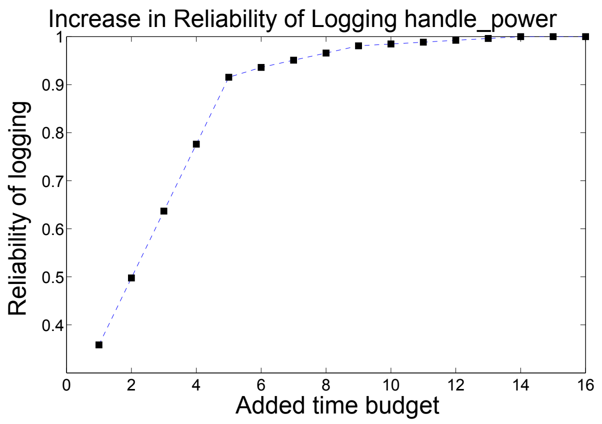


Fig. 6. Effect of increasing the time budget for logging in handle_power.

X. Case Study II: Flash Filesystem

In this section, we describe the flash filesystem implementation. This second case study confirms the observations from the first case study and provides more evidence for our conclusions.

A. Overview

We investigated an implementation of a wear-levelling FAT-like filesystem for flash devices [15]. The code was originally written by Hein de Kock for 8051 processors. We slightly modified the original implementation so that it would compile with `sdcc`; in particular, we needed to modify the header files to get the code to compile. The implementation consists of about 3000 non-blank, non-comment lines of C code. We ran our tool on 30 functions from the `fs.c` file, dropping some uninteresting functions with mostly straight-line control-flow. Of the 30 functions, 4 functions had more than 100 basic blocks, and `fclose` had 200 basic blocks. For this case study, we also assume that the time budget is the execution time of the longest running path in the function, $o_f = 0$, $o_c = 1$, and no interrupts.

B. Measurements

Figure 7 compares density functions for four procedures in the filesystem implementation, both before and after instrumentation. The solid blue line represents the density function of the original procedures, while the dashed red line represents the density function for the instrumented versions. Each of this figures clearly shows that the original idea underlying our method of time-aware instrumentation, as outlined in Figure 1, works well.

The procedure *fsetpos* shown in Figure 7(b) exhibits the biggest difference between instrumented and non-instrumented versions. The reason is that although this procedure contains many assignments spread across different paths, most assignments do not lie on the worst-case path. The instrumentation engine can therefore capture assignments along these non-critical paths, raising their execution time and putting them closer to the execution time of the worst-case path. Since the engine can capture assignments on many paths, the density function of the execution time for the instrumented version shows a large increase on the right part of the figure, along with a steep decrease on the left part of the figure.

The procedure *rename* shown in Figure 7(d) demonstrates that sometimes the developer might want to add time to the budget for instrumenting to enable the instrumentation of the worst-case path. Figure 8 shows that even with a small increase in the time budget, the reliability can increase significantly. Figure 7 shows the function *fputs* without any additional increase in the time budget, Figure 8(a) shows the function with an extra budget of three assignments, and Figure 8(c) shows the function with an extra budget of 15 assignments. Figure 8(d) summarizes how instrumenting *fputs* improves as we add more time to the time budget for the instrumentation.

XI. Related Work

Debugging embedded systems is typically achieved through capture and replay approaches. Capture and replay approaches include tracing. Generally, a program is instrumented to generate traces which are then replayed offline, potentially in a simulator. Capture and replay [16] is a well-established method for debugging concurrent and distributed systems going back to early publications in 1987 [17]. Thane et al. [18], [19], [18] propose a software-based approach for monitoring and replay in distributed real-time systems. Other approaches concentrate on the problem of debugging concurrent programs [20], [21]. However, the mechanisms used for instrumentation in these systems do not consider their impact on the timing of the application, which is the main aim of this work.

Tsai et al. [22] propose a monitoring approach that minimizes the probe effect by using additional hardware. Our proposed approach relies only on software mechanisms. Dodd et al. [23] propose a software-based approach targeted for multiprocessor machines which uses software instruction counters. In this approach, the program execution is cleverly distributed to two processors to minimize the probe effect. Our approach targets microprocessor systems which only have a single execution unit.

Other monitoring approaches include AspectJ [1], Etch [2] and Valgrind [24]. AspectJ is an implementation of aspect-oriented programming, which enables developers to execute given code when certain events occur. AspectJ supports instrumentation since potential events include memory writes; however, AspectJ will instrument these events indiscriminately, without respect to resource bounds. Etch is a static monitoring and instrumentation framework for instrumenting Win32/Intel executables; it could support a time-aware instrumentation plugin. Valgrind is a dynamic monitoring framework which has been successfully used for detecting problematic memory accesses. All of these monitoring approaches are for non-real-time applications running on desktop computers; to our knowledge, there are no proposed instrumentation approaches for embedded systems.

Another category of tools are dynamic instrumentation tools like DynamoRIO [25] and Pin [26]. Dynamic instrumentation tools rewrite instructions executed at run time based on a instrumentation specification. Our tool work uses static instrumentation as it provides better estimates of how the system

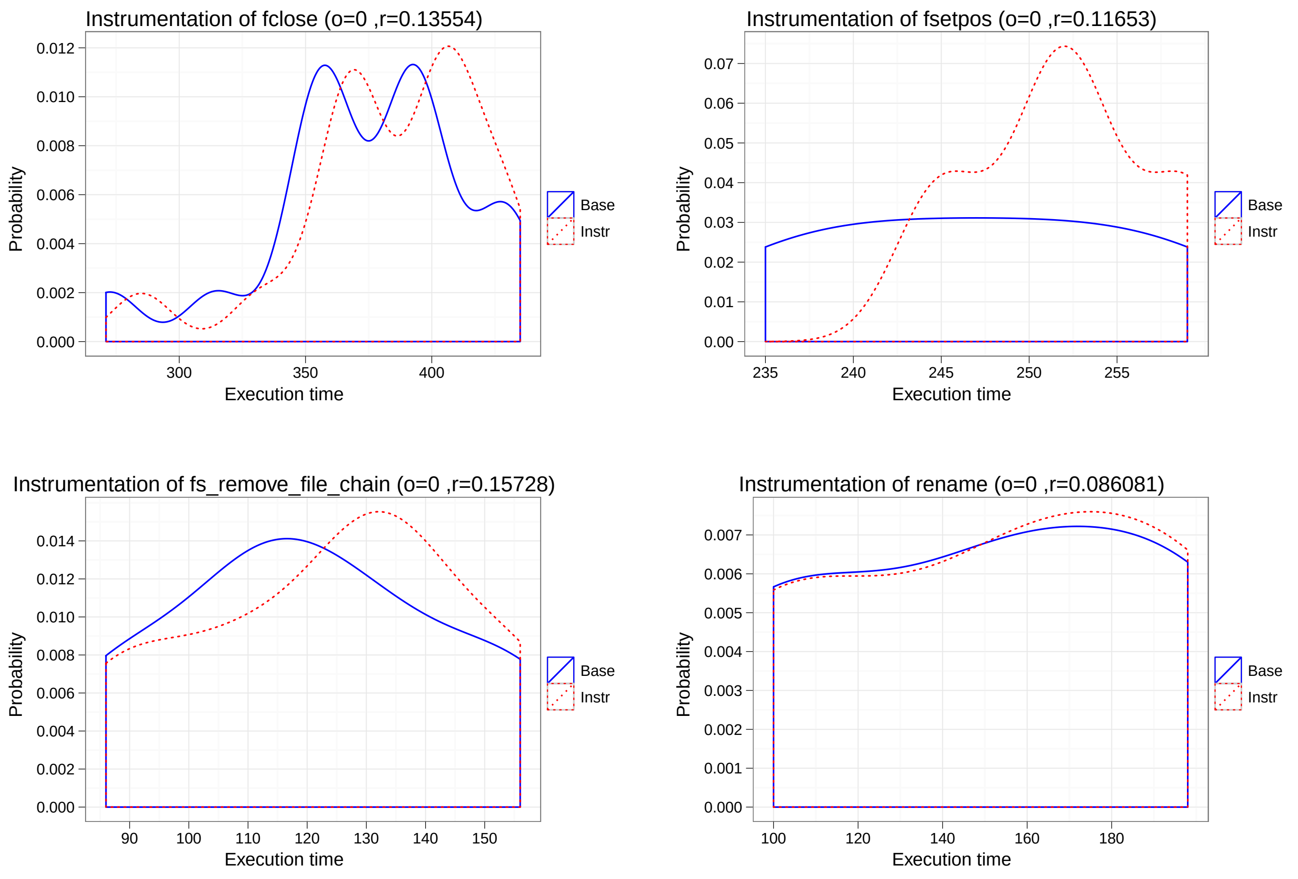


Fig. 7. Examples in instrumenting functions in the filesystem implementation.

will behave at run time, which is important for safety and time-critical applications.

Research on static instrumentation has culminated in different approaches and tools including Executable Editing Library (EEL) [27], Etch [28], Morph [29], and Atom [30]. Each of these tools can be used to instrument an application to, for instance, trace assignments; however, none of them consider timing constraints or work in the context of real-time embedded systems.

XII. Discussion

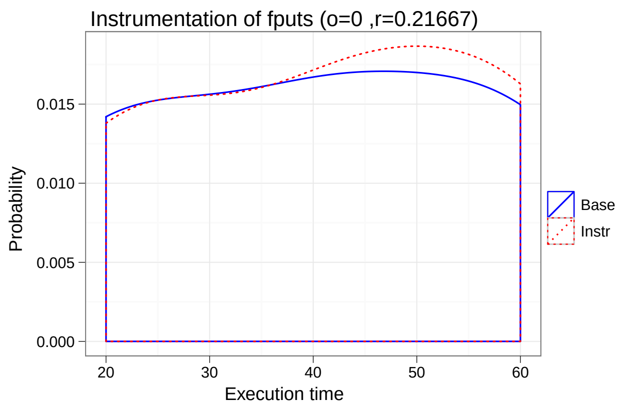
Our case studies with the OpenEC keyboard controller and flash filesystem implementation provided interesting insights into our proposed solution for smart instrumentation. In this discussion, we address questions about the general approach of smart instrumentation and lessons learnt.

Partial instrumentation, as a special type of ensuring timeliness, is useful for building an inductive debugging mechanism for deployed resource and space constrained systems. It is hard to reproduce bugs from user reports [31] and thus having even a partial trace can help without additional tool support. In addition, the partial trace then can be input for additional debugging tools [32], [33]. Note, however, that the achieved reliability of the partial instrumentation depends on the execution paths taken at run time.

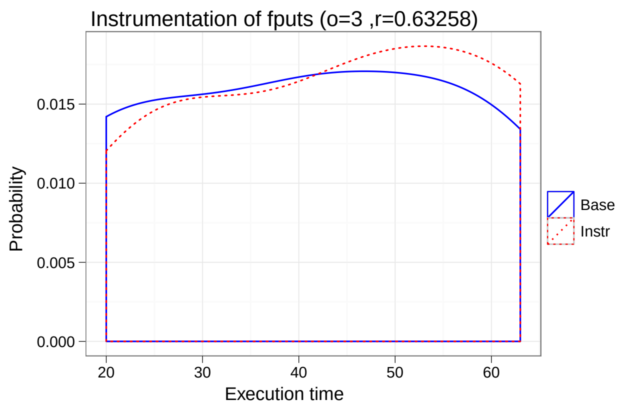
A developer may require definitive information about a particular assignment to a variable. In such a case, the developer must force the instrumentation point at the cost of (a) lowering overall reliability and (b) possibly violating a temporal constraint. Our framework allows the developer to estimate the impact on both of these costs and make an informed decision. Similarly, a developer may define instrumentation-free sections in the code, for example time-sensitive operations outside the worst-case path. Both forcing and exclusion are trivial extensions to the presented work and target usability.

The anlaysis we require prior instrumenting the source code is executed offline and can be done on a per-function basis. While the run time complexity of the analysis is exponential, our method aims at background/foreground systems with a reasonable code base as usually found in microcontroller systems. The largest example we tested contained about 124 000 nodes and using Matlab's ILP solver required less than a second to solve this on a standard dual-core laptop with two GB of memory.

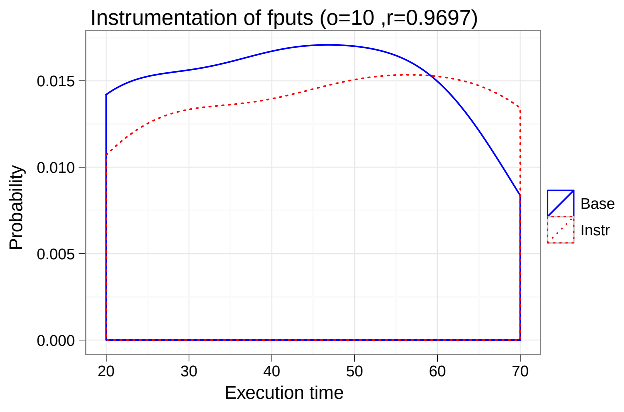
The calculation of the instrumentation reliability shown in Eq (2) is off by a small factor if the developer uses the framework to extract control information. However, if all paths except one—the worst-case path—contain at least one instrumentation point, then if the run returns no data, the developer knows that the program executed the worst-case path. In this specific case, the reliability can be increased



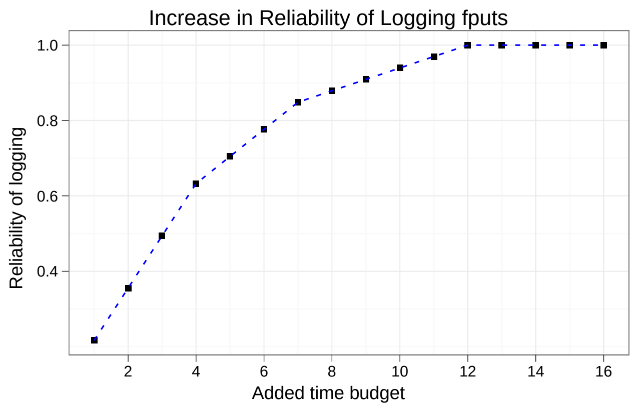
(a) Without increased time budget.



(b) With a three assignments added to the time budget.



(c) With a 15 assignments added to the time budget.



(d) Increase in reliability.

Fig. 8. Examples of increasing the reliability by increasing the time budget.

by the information gained times the likelihood of taking the worst-case path. This is a very rare but notable case which Eq (2) ignores.

Another extension of our work is to emit the line number in which the variable assignment too place together with the variable assignment value. This allows the developer to recompute the accuracy of the trace after the logger captured the trace. The work in Section VI helps the developer to estimate the accuracy of the trace before actually running the program. If the reliability is too low, then the developer might want to add time to the time budget for tracing.

XIII. Conclusion

We have proposed *time-aware instrumentation*, a novel approach to program instrumentation. The idea of time-aware instrumentation applies to a variety of properties and in this work we showed how it can be used to maximize trace reliability and computing the minimal trace buffer size. Our approach enables developers to, among other applications, follow the evolution of program variables over the course of a program's execution.

We have evaluated our time-aware instrumentation approach by automatically creating models of the OpenEC keyboard controller code and a flash filesystem, and running simulations on our models. Our approach successfully shifts the distribution of runtimes to more effectively use a time budget without exceeding it, and enables calculations of the additional logging reliability available with small violations (or increments) of the time budgets.

Acknowledgements

We thank Frieder Ferlemann from the OpenEC project for the help with the source code. We also thank the reviewers for their feedback and suggestions. This research was supported by the Natural Sciences and Engineering Research Council of Canada.

Sebastian Fischmeister is Assistant Professor with the Department of Electrical and Computer Engineering at the University of Waterloo in Canada. He received the Dipl.-Ing. degree in Computer Science at the Vienna University of Technology, Austria, and his PhD degree in Computer Science at the University of Salzburg, Austria. His primary research interests lie at the intersection of software systems, embedded networking, and applications of formal methods in the context of real-time embedded systems.

Patrick Lam Patrick Lam is an Assistant Professor in the Department of Electrical and Computer Engineering at the University of Waterloo. He completed his PhD at the Massachussetts Institute of Technology. His primary research interests include applications of program analysis techniques to software engineering, particularly to program verification and understanding.

References

[1] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *European Conference on Object-oriented Programming*, ser. Lecture Notes in Computer Science, J. L. Knudsen, Ed., vol. 2072. Springer, 2001, pp. 327–353.

[2] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen, "Instrumentation and optimization of Win32/Intel executables using Etch," in *Proceedings of the USENIX Windows NT Workshop 1997*, 1997, pp. 1–8.

[3] J. J. Labrosse, *MicroC OS II: The Real Time Kernel*. CMP Books, 2002.

[4] B. Dobbing and A. Burns, "The Ravenscar Tasking Profile for High Integrity Real-time Programs," in *Proceedings of the 1998 annual ACM SIGAda international conference on Ada (SIGAda)*. New York, NY, USA: ACM, 1998, pp. 1–6.

[5] S. Fischmeister and I. Lee, *Handbook on Real-Time Systems*, ser. Information Science Series. CRC Press, 2007, ch. Temporal Control in Real-Time Systems: Languages and Systems, pp. 10–1 to 10–18.

[6] M. Li, T. V. Achteren, E. Brockmeyer, and F. Catthoor, "Statistical Performance Analysis and Estimation of Coarse Grain Parallel Multimedia Processing System," in *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 277–288.

[7] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The Worst-case Execution-time Problem—Overview of Methods and Survey of Tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[8] "OpenEC Project Site," Web site, 2008, http://wiki.laptop.org/go/OpenEC.

[9] "SDCC - Small Device C Compiler," URL http://sdcc.sourceforge.net/, 2008.

[10] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "applying New Scheduling Theory to Static Priority Pre-emptive Scheduling," *Journal of Software Engineering*, vol. 8, no. 5, pp. 284–292, 1993.

[11] J. Liu, *Real-Time Systems*. New Jersey: Prentice-Hall, 2000.

[12] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[13] S. Dieckmann and U. Hölzle, "A study of the allocation behavior of the SPECjvm98 Java benchmarks," in *Proceedings of ECOOP 1999*, vol. LNCS, 1999, pp. 92–115.

[14] R. M. Karp, "Reducibility Among Combinatorial Problems," in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds. New York: Plenum Press, 1972, pp. 85–103.

[15] H. de Kock, "small-ffs," http://http://code.google.com/p/small-ffs, September 2009.

[16] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere, "JaRec: a portable record/replay environment for multi-threaded Java applications," *Software—Practice & Experience*, vol. 34, no. 6, pp. 523–547, 2004.

[17] T. Leblanc and J. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *Transactions on Computers*, vol. C-36, no. 4, pp. 471–482, Apr. 1987.

[18] D. Sundmark, H. Thane, J. Huselius, and A. Pettersson, "Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies," in *Proceedings of the 5th International Workshop on Algorithmic and Automated Debugging (AADEBUG03)*, Gent, Belgium, Sept. 2003, pp. 211–222. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=0573

[19] H. Thane, "Monitoring, Testing and Debugging of Distributed Real-Time Systems," Ph.D. dissertation, Department of Computer Science and Electronics, Mälardalens University, May 2000. [Online]. Available: http://www.mrtc.mdh.se/index.php?choice=publications&id=0242

[20] M. Ronsse, K. De Bosschere, M. Christiaens, J. de Kergommeaux, and D. Kranzlmüller, "Record/replay for nondeterministic program executions," *Communications of the ACM*, vol. 46, no. 9, pp. 62–67, 2003.

[21] K. Audenaert and L. Levrouw, "Interrupt Replay: A Debugging Method for Parallel Programs with Interrupts," *Microprocessors and Microsystems*, vol. 18, no. 10, pp. 601–612, Dec. 1994.

[22] J. J. P. Tsai, K.-Y. Fang, H.-Y. Chen, and Y.-D. Bi, "A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 897–916, 1990.

[23] P. Dodd and C. Ravishankar, *Monitoring and debugging distributed real-time programs*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1995, ch. Monitoring and debugging distributed real-time programs, pp. 143–157.

[24] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2007, pp. 89–100. [Online]. Available: http://portal.acm.org/citation.cfm?id=1250734.1250746

[25] 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: IEEE Computer Society, 2003, pp. 265–275.

[26] 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 *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, 2005, pp. 190–200.

[27] J. R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI)*. New York, NY, USA: ACM, 1995, pp. 291–300.

[28] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, and B. Bershad, "Instrumentation and optimization of Win32/Intel executables using Etch," in *Windows NT Workshop*, 1997.

[29] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith, "System Support for Automatic Profiling and Optimization," in *Proceedings of the sixteenth ACM Symposium on Operating Systems Principles (SOSP)*. New York, NY, USA: ACM, 1997, pp. 15–26.

[30] A. Srivastava and A. Eustace, "ATOM A System for Building Customized Program Analysis Tools," *SIGPLAN Not.*, vol. 39, pp. 528–539, April 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=989393.989446

[31] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What Makes a Good Bug Report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. New York, NY, USA: ACM, 2008, pp. 308–318.

[32] 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 (CGO)*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 221–230.

[33] 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 (OOPSLA)*. New York, NY, USA: ACM, 2007, pp. 535–552.