In Proc. of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS),
San Francisco, United States, April 2009.
Revised version.

On Time-Aware Instrumentation of Programs

Sebastian Fischmeister
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
sfischme@uwaterloo.ca

Patrick Lam
Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Canada
p.lam@ece.uwaterloo.ca

***Abstract*—Software instrumentation is a key technique in many stages of the development process. It is of particular importance 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 the worst case, this perturbation leads to new system behaviours that prevent the developer from locating the original misbehaviours.**

In this work, we propose an instrumentation technique for applications with temporal constraints, specifically targetting background/foreground systems. Our framework permits reasoning about space and time for 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 for code size and time budgets. Finally, we apply the theory to a concrete case study and instrument the OpenEC firmware for the keyboard controller of the One Laptop Per Child project.

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

I. Introduction

Instrumentation and tracing are a key activity in debugging microcontroller-based embedded systems. The instrumented program produces data traces which the developer uses to locate the origin of a misbehavior in the system under test. For example, if the trace shows incorrect control flow at a conditional branch, then the branching conditions or the input values will most likely contain 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. In the worst case, the instrumentation overhead introduces Heisenbugs [1]. Experienced systems developers know that sometimes a *nop* instruction at the right place magically solves problems. In time-related Heisenbugs, the instrumentation overhead performs this magical transformation, so that the developer cannot locate a misbehavior in the instrumented program which clearly exists in the original. To minimize the chances of producing a timing-related Heisenbug, the instrumentation should consider time budgets and have minimal overhead.

Related works on instrumentation have 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 [2] and Etch [3] 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.

This work concentrates on time-aware instrumentations of background/foreground systems. This structure dominates microcontroller systems due to its structure and efficient resource utilization [4], [5]. Technically, background/foreground systems 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. The foreground task preempts the background task whenever a serviceable interrupt line becomes asserted. The background task never preempts the foreground task. We assume no nested interrupts and one interrupt priority level, yet our results still hold for the target class of systems.

II. Rationale

The key idea behind *time-aware instrumentation* of a system is to transform the execution-time distribution, maximizing the reliability of the trace while always staying within the time budget. With reliability we mean that the instrumentation provides useful data over a longer period time of tracing. A time-aware instrumentation injects code, potentially extending the execution time on all paths, and ensures that no path takes longer than the specified time budget. For a hard real-time system, there may be no slack and therefore this time budget may be zero, yet the developer may still trace the execution in paths other than the one with the worst-case timing. Figure 1 shows the expected consequences of time-aware instrumentation on the probability density function of a loop's execution time. The x-axis specifies the required execution time of the loop. The y-axis indicates the frequency

This research was partly sponsored by NSERC DG 357121-2008 and by CIMIT under U.S. Army Medical Research Acquisition Activity Cooperative Agreement W81XWH-07-2-0011. The information contained herein does not necessarily reflect the position or policy of the Government, and no official endorsement should be inferred.

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 the algorithm instruments paths and increases their running times, but instruments them so their execution times never pass the deadline. Note that this model concentrates on acquiring data from the functions. Another problem is how to transport the data from the chip to an external analysis unit. While that problem requires detailed study, a simple solution commonly found is piggy packing the buffer information on serial or network communication.

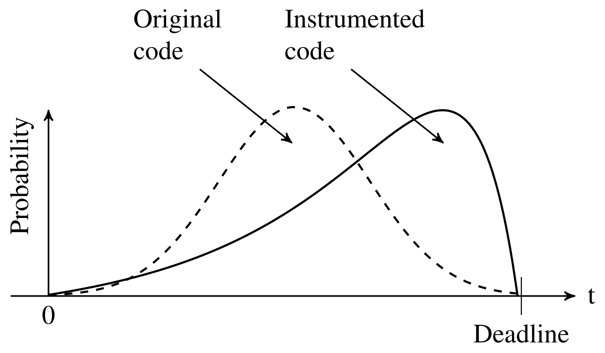

Fig. 1. Execution-time distribution before and after time-aware instrumentation showing the shift in the expected execution time.

So, what do we need to perform time-aware instrumentation? First, we need an underlying model to abstract properties from the source code. Since we concentrate on timed systems, this model should include the temporal behavior and the control flow, together with what data needs to be logged. This model 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 for the instrumentation overhead, the system provides the best instrumentation possible in terms of trace reliability.

The contributions of this paper include:

- a definition for instrumentation reliability;
- a definition of "time-aware instrumentation", which instruments 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
- results exploring the impact of time-aware instrumentation on the OpenEC keyboard controller code.

III. 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 VII-D 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 VIII) 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 a first try, 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 instrumentation is too low, then we either will have to give up, if we cannot extend the time budget available for the function and the instrumentation, or we can extend the time budget and thereby increase the reliability. 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 execution paths to minimize the required code size. Afterwards we can recompile the program and collect traces from the instrumentation.

IV. 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

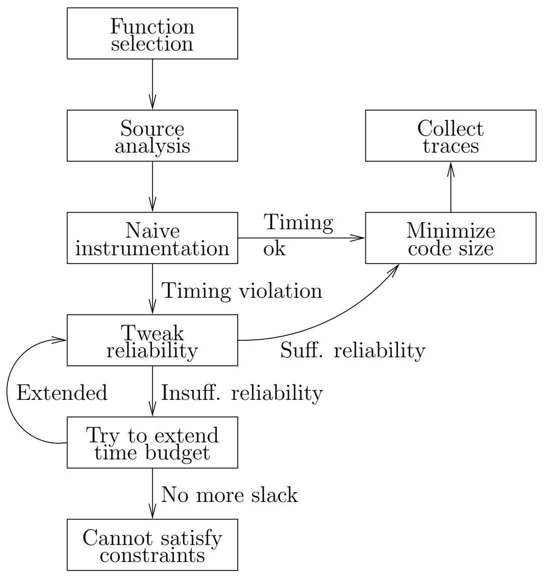

Fig. 2. Workflow of applying time-aware instrumentation.

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.

```
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; Section VI discusses this issue in more detail.

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 a snippet of instrumentation code after `movx r0, a`. With time-aware instrumentation, we compute the cost of the procedure and instrument it, if our budget allows. If the budget does not allow for complete instrumentation, we instrument a subset of the writes to memory, maximizing reliability and optimizing 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.

V. 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, G contains a large cycle ("super loop"), representing the forever-running background task, with background subtasks on the spine of the large cycle.

An *instrumentation* of a software program is the insertion of custom code at specific insertion points into a program. An *instrumentation operation* is the piece of code that realizes the desired 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, to be read once the program terminates or at the bottom of the super loop.

A. Static Analysis Approach

We have built a static analysis tool which accepts background/foreground implementations 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.

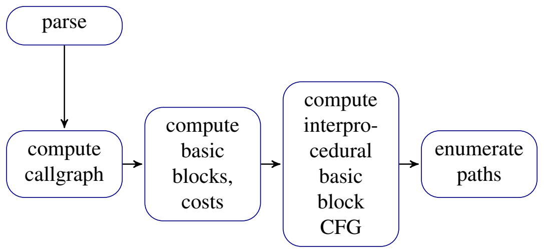
parse
compute callgraph
compute basic blocks, costs
compute interprocedural basic block CFG
enumerate paths

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

We chose to analyze assembler code directly. Our case study was written for an 8051-family microcontroller, which is simple to model: it is sufficient to count the (constant) number of cycles each instruction takes to execute. We had considered analyzing the C source code instead. While the control structure of the program is immediately visible in the C source code, it is generally possible to reconstruct a program's structure from assembler code [9], especially compiler-generated assembler code. 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 first parses the assembler code emitted by SDCC [10], the C compiler for the OpenEC project. We used a heuristic to estimate the extent of each procedure, since this information is not explicit in the assembler code: while the start of each procedure is labelled, the end is not. We define the contents of a procedure to be the instructions between the start of a procedure and the start of the subsequent procedure. We also ignore all instructions that do not belong to a "CSEG" code segment (for instance, initialization code belongs to a "GSINIT" global initialization segment). It is straightforward to transform and unparse the assembler code to produce instrumented applications.

Next, we compute the call graph and basic blocks. Our target class of embedded programs generally is free of function pointers or dynamic dispatch, we were able to use a straightforward callgraph construction algorithm. We also used standard algorithms to divide procedures into basic blocks, starting a new block whenever a statement has more than one predecessor or successor, or is a call statement. We then computed 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 up a 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 background/foreground implementations consist of a main "super loop" which executes forever; 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 from one (fixed-size) array buffer to another 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 logged variables L. The function $c : V \rightarrow \mathbb{R}$ specifies the required computation time c, or cost, for vertex v. So, $c(v_0) = 12.2t$ means that the basic block at vertex v_0 requires 12.2 time units for its execution. Edges $e := \langle v_s, v_d\rangle$ specify transitions from source vertex v_s to destination vertex v_d. The function $p : E \rightarrow [0, 1]$ computes 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 functions c and p. Microcontroller vendors typically provide cycle accurate simulators which allow the developer to measure the execution time of the 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]. Our analysis framework helps developers compute c by providing the cycle counts for each basic block. We expect that the developer will estimate p by collecting profiling data.

VI. Accommodating 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 always eventually terminate. We can then adjust the execution time of any measurement c to:

$$c'(x) = c(x) + \sum_{irq} (\lceil c(x) f(irq_i) \rceil c(irq_i)) \tag{1}$$

with $c(irq_i)$ as the execution time of the interrupt and $f(irq_i)$ as its minimal inter-arrival rate frequency. 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 following equation computes the worst-case instrumentation overhead o. If o is less or equal to the specified time budget, then instrumentation will be viable.

$$o = \max(\sum_{v \in p} c'(v.L)) \text{ for all } p \in P \tag{2}$$

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 3 125ns. The UART interrupt arrives with a frequency of 11 520Hz, resulting in a worst-case overhead for the interrupt of 36ms/s of execution time. Thus, for a basic block with an execution time of 10ms, we compute $c'(x) = 10.3125$ms.

VII. 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 path with the 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 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 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 reliability, this means 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

Algorithm 1 calculates the instrumentation reliability for a runtime-tracing insertion point using a depth-first search algorithm. Recall that $v.A$ represents the set of all assignments (and hence all insertion points) at v, while $v.L$ represents the set of monitored assignments at v. In the recursive function *hit()*, the algorithm traverses along a branch until (a) it detects that a vertex logs the assignment or (b) the variable gets reassigned. If the branch results in monitoring (case (a)), the algorithm will add to the computed reliability the probability of the program taking that particular path. If the branch misses (case (b)), the algorithm will add zero to the probability for the recursive case's return value. This algorithm essentially implements a depth-first search and all standard complexity analysis results apply to it.

Algorithm 1 Calculate reliability for a single assignment.

```
Require: starting $v : v.A \Leftarrow v.A \setminus x$, start with $p = 1$
  procedure hit($v, x, p$)
    if $x \in v.A$ then return 0 end if
    if $x \in v.L$ then return $p$ end if
    $a \Leftarrow 0$
    for all $e$ such that $e.v_s = v$ do
      $a \Leftarrow a + \text{hit}(e.v_d, x, p \cdot t.p)$
    end for
    return $a$
  end procedure
```

B. Reliability For A Path

The instrumentation reliability of a path p, denoted $r(p)$, equals 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 $v.L$ can contain entries that are only relevant to other paths.

Algorithm 2 shows how we calculate the reliability for a path in a single pass. Essentially, as the algorithm visits a path, it remembers the variables and then adjusts one counter for misses and one for hits accordingly. The operator "next" will point to the next element in the path, if there exists such an element; otherwise, it will point to "nil". We use variable p to iterate over the path's vertices. The set l contains all variables that need to be monitored.

Algorithm 2 Calculate the reliability of logging for a path.

```
$p \Leftarrow v_0$
$l \Leftarrow \emptyset$
**while** $p \neq$ nil **do**
  **for all** $x \in p.A$ **do**
    **if** $x \in l$ **then**
      $\text{miss} \Leftarrow \text{miss} + 1$
    **end if**
  **end for**
  **for all** $x \in p.L$ **do**
    **if** $x \in l$ **then**
      $\text{hit} \Leftarrow \text{hit} + 1$
      $l \Leftarrow l \setminus x$
    **end if**
  **end for**
  $l \leftarrow l \cup p.A$
  $p \Leftarrow \text{cur}.\text{next}$
**end while**
$\text{miss} \Leftarrow \text{miss} + |l|$
```

C. Reliability For Instrumentations

The *instrumentation reliability of a partial instrumentation*, $r(P)$, is the sum of the weighted reliability of all possible paths using the path probabilities as weights. $p(p) = \prod_{e \in p} p(e_i)$ gives the probability of taking a specific path. (In multiplying the probabilities together, we assume that they are independent. Many systems, including SPIN [12], assume independent probabilities at different choice points, as we do here.)

$$r(P) = \sum_{p_i \in P} r(p_i) p(p_i) \quad (3)$$

D. 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 formulate the problem as a linear programming problem. Equation (4) shows the function to be maximized. Variables x_i store the value of the insertion point (i.e., the number of trace variables in the basic block v_i). If paths share basic blocks (vertices), then the optimization function adds up the coefficients of the x_is to obtain path costs that respect sharing. The function $p(p_i, v_j)$ gives the probability that path p_i will eventually hit basic block v_j. For soft real-time systems, the values can be obtained by run-time analysis. Safety-critical systems require static analysis and reduce the problem to a binary linear programming problem. Inequalities (5) and below represent the problem constraints: the total instrumentation overhead (probability of reaching the basic block multiplied with whether that variable is of interest [0 or 1] multiplied with the instrumentation overhead for the variable in that basic block) must be less than the time budget tb. The term $tb - \sum_{v \in p_j} c'(v_i)$ computes the available time budget for the instrumentation (in real-time systems tb is known as slack time). Increasing the number of insertion points (i.e., setting x_i non-zero) can increase the execution time on each path. Finally, constraints (6) list the boundary conditions and limit the number of variables per basic block.

$$\max \sum_{p \in P} \sum_{v \in p_i} p(p_i, v_i) x_i \quad (4)$$

$$\sum_{v \in p_0} p(p_0, v_i) \cdot x_i \cdot c(x_i) \leq tb - \sum_{v \in p_0} c'(v_i) \quad (5)$$

$$\cdots$$

$$\sum_{v \in p_n} p(p_n, v_i) \cdot x_i \cdot c(x_i) \leq tb - \sum_{v \in p_n} c'(v_i)$$

$$x_0 \leq |v_0.A| \quad (6)$$

$$\cdots$$

$$x_n \leq |v_n.A|$$

Note that in a safety-critical system with zero time budget for overhead, to calculate the worst case, the function $p(p_i, v)$ will always return one if the block is reachable and otherwise zero regardless of the probability of reaching the value. Furthermore, in hard real-time systems, the time budget tb is usually a task's maximal response time.

VIII. 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], [3], [2], do not use the minimal number of insertion points in general.

Unfortunately, finding the minimal number of insertion points is NP-complete. We can show this by reducing the instrumentation problem to the NP-complete hitting set problem [14]. First, we extract the control flow graph with the variable assignments and then convert it into static single assignment form. The left part of Figure 4 shows the control flow graph (CFG) with four assignments to variable x. To calculate the minimal number of necessary insertion points, we compute the hitting set shown in the right part of Figure 4. A line between an assignment x and a basic block b means that the assignment can be captured in this basic block. Essentially, we need to compute the minimal subset of basic blocks on the right that covers all variables on the left.

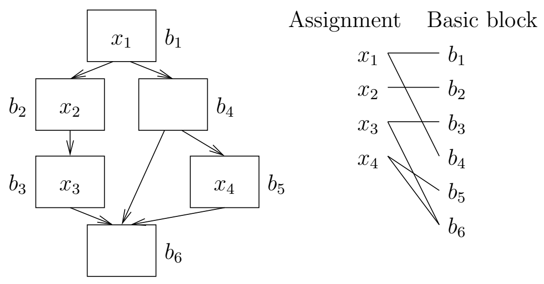

Fig. 4. Minimizing instrumentation points.

Proposition 1: 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: instrumenting a program can only increase the code size. Therefore, the program with the fewest insertion points also has the smallest code size.

a) Towards the minimal instrumentation.: Our goal is to transform the naive instrumentation with maximal reliability so as to preserve reliability and reduce the size of the instrumentation. However, we also must ensure that the minimization does not change the time budget of the non-optimized instrumentation. Therefore, we 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 has the same set of executions as the original instrumentation. Clearly, it is necessary for transformations to be semantics-preserving if they are to preserve reliability. An example of a semantics-preserving transformation is one that delays recording a variable, but not beyond a subsequent write of that variable. A decreasing transformation does not increase the number of insertion points in an instrumentation. Being decreasing is a sufficient condition for a transformation to ensure that the transformed instrumentation does not exceed its time budget if the original instrumentation did not exceed its 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.

IX. 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 11, 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 $b_{max} = \max(s(p_i))$ for all $p_i \in P$. The *expected* buffer size is $b_{exp} = \sum_{p_i \in P} s(p_i)p(p_i)$.

X. OpenEC Case Study

To demonstrate the effectiveness of our approach, we trace the handle_power function of the 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:

- Tolerating zero overhead, with what reliability can we monitor variables in this function without breaking the temporal bound? 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 question, we implemented (a) a static analysis tool outlined in Section V-A in OCaml and (b) the ILP problem from Section VII-D in Matlab.

A. Trace Reliability for OpenEC

Figure 5 presents the trace reliabilities along the different paths using a zero overhead time budget. 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 VII-C and our abstraction, the monitoring reliability of the handle_power is 35.83% for this scenario.

B. Execution Time

Figure 6 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 Section II 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

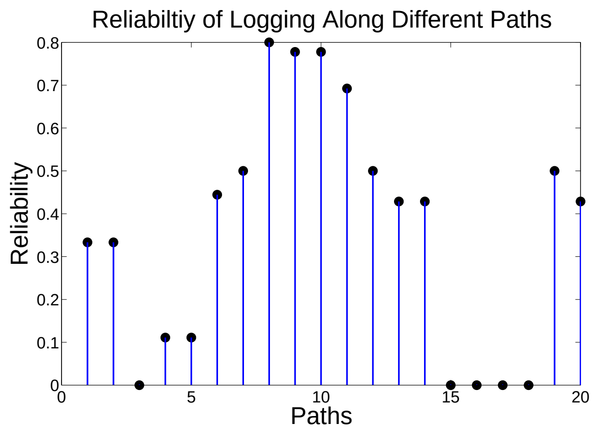

Fig. 5. Monitoring reliability of function handle_power in the 'all out' scenario.

assignments to heap variables outside the worst-case path. More assignments per basic block imply a more prominent shift in the density function.

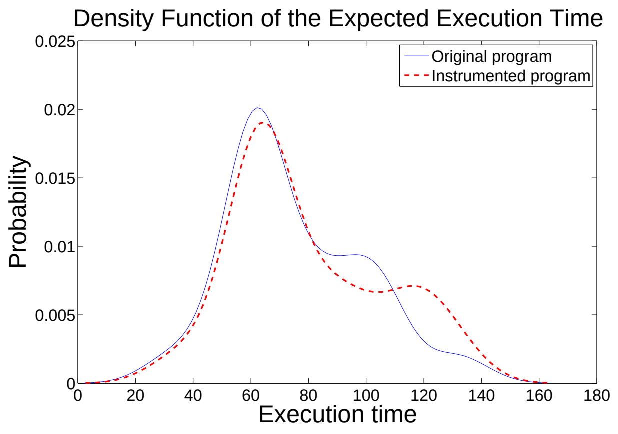

Fig. 6. Shift in the density function of the execution time of in the function handle_power.

C. Minimal Buffer Size

To calculate the minimal buffer size, we use Algorithm 2 with the modifications described in Section IX. The minimal-required buffer size is independent of any particular path's tracing reliability, because 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 devote a time budget to the tracing effort; consider a heavily-loaded system that drives a motor. The motor may tolerate jitter in its duty cycle, however, reliable operation demands as little jitter as possible. During the monitoring effort, the system might 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 7 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.

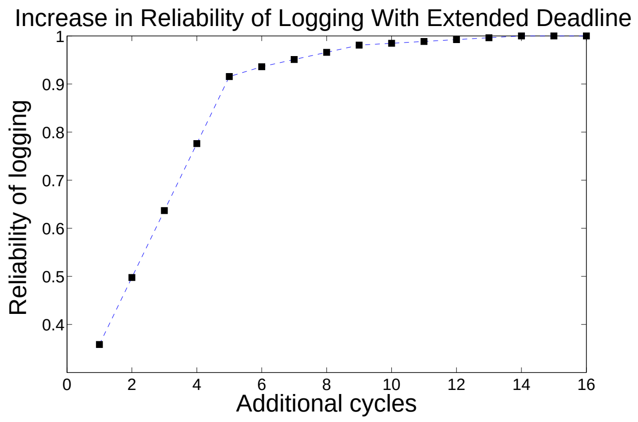

Fig. 7. Effect of increasing time budget for logging in handle_power.

XI. Related Work

Debugging embedded systems is typically achieved through capture and replay approaches; such approaches include tracing. In such approaches, the program is instrumented to generate traces which are then replayed offline, potentially in a simulator. Capture and replay [15] is a well-established method for debugging concurrent and distributed systems going back to early publications in 1987 [16]. Thane et al. [17], [18], [17] propose a software-based approach for monitoring and replay in distributed real-time systems. Other approaches concentrate on the problem of debugging concurrent programs [19], [20]. 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. [21] propose a monitoring approach that minimizes the probe effect by using additional hardware. Our proposed approach relies only on software mechanisms. Dodd et al. [22] 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 aims for microprocessor systems which only have a single execution unit.

Other monitoring approaches include AspectJ [2], Etch [3] and Valgrind [23]. 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.

XII. 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 extracting models of the OpenEC keyboard controller code and running simulations on our model; we observed that it successfully shifts the distribution of runtimes so as to more effectively use the available time budget without exceeding it, and calculated the additional logging reliability which can be obtained by small violations 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.

References

[1] M. Winslett, "Bruce Lindsay speaks out: on System R, benchmarking, life as an IBM fellow, the power of DBAs in the old days, why performance still matters, Heisenbugs, why he still writes code, singing pigs, and more," *SIGMOD Rec.*, vol. 34, no. 2, pp. 71–79, 2005.

[2] 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* (J. L. Knudsen, ed.), vol. 2072 of *Lecture Notes in Computer Science*, pp. 327–353, Springer, 2001.

[3] 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*, pp. 1–8, 1997.

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

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

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

[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] J. Miecznikowski and L. Hendren, "Decompiling Java bytecode: Problems, traps and pitfalls," in *Proceedings of Compiler Construction 2002*, vol. 2304 of *Lecture Notes in Computer Science*, pp. 111–127, 2002.

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

[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, pp. 92–115, 1999.

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

[15] 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.

[16] T. Leblanc and J. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *Transactions on Computers*, vol. C-36, pp. 471–482, Apr. 1987.

[17] D. Sundmark, H. Thane, J. Huselius, and A. Pettersson, "Replay Debugging of Complex Real-Time Systems: Experiences from Two Industrial Case Studies," in *Proc. of the 5th International Workshop on Algorithmic and Automated Debugging (AADEBUG03)*, (Gent, Belgium), pp. 211–222, September 2003.

[18] H. Thane, *Monitoring, Testing and Debugging of Distributed Real-Time Systems*. PhD thesis, Department of Computer Science and Electronics, Mälardalens University, May 2000.

[19] 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.

[20] K. Audenaert and L. Levrouw, "Interrupt Replay: A Debugging Method for Parallel Programs with Interrupts," *Microprocessors and Microsystems*, vol. 18, pp. 601–612, 12 1994.

[21] 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.

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

[23] 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), pp. 89–100, ACM, 2007.