Program Transformation for Time-Aware Instrumentation

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

***Abstract*—Instrumentation is a valuable technique to gain insight into a program's behavior. Safety-critical real-time embedded applications are time sensitive and so instrumentation techniques for this domain must especially consider timing.**

This work establishes the basis for measuring the effectiveness of approaches for time-aware instrumentation in addition to coverage. We define the *ETP shift effectiveness* metric and define its optimality criterion. We identify locations in the program where program transformation techniques can be applied to increase the instrumentability of the program. We subsequently use the proposed metric to evaluate two transformation methods that improve the effectiveness and coverage of current techniques for time-aware instrumentation by a factor of five.

I. Introduction

Tracing a program usually means extracting information from the program while it runs. Depending on the method, this extraction process generally causes perturbation in the application. The perturbation originates from the instrumentation process. Software instrumentation techniques [1], [2], [3] insert tracing code into the original program code. Usually, the more tracing code the program executes during the run, the more the slow down and perturbation. The reason is that, in general, the addition of more tracing code increases the number of instructions the processor executes thus leading to a longer execution time. Dynamic instrumentation approaches [4], [5] modify the binary at runtime and thus cause highly non-deterministic timing behavior. Hardware-supported tracing [6], [7] use special hardware interfaces to stream data off chip. Even this method can cause significant perturbation [8].

Safety-critical real-time applications must be correct and meet timing requirements. Timing perturbations are thus particularly harmful for such applications. Therefore, prior work investigates mechanisms to instrument programs without affecting their timing constraints and functional behavior.

This work specifically investigates time-aware instrumentation [9]. This approach tries to preserve logical correctness as well as meeting timing constraints. Often, this means instrumenting only on non-worst-case paths of the program. While a minor influence maybe be acceptable due to a specified timing constraint for debugging, naive instrumentation will usually violate such constraints. Time-aware instrumentation attempts to honor the timing constraint and shifts the execution time profile (ETP) closer to the programs deadline (without exceeding the debugging constraint).

Case studies investigated in related work [9] demonstrate the promise of the general concept but the results revealed new problems. Figure 1 shows the execution time profile of a case study reported in [9] on the OLPC keyboard controller. The figure shows the success of time-aware instrumentation in shifting the ETP of the instrumented program. While the shift in the execution time profile is visible, it is lower than expected. The expectation was a larger shift in the time profile towards longer execution times. Further investigation revealed that, in this example, about 25 percent of the paths share basic blocks with the worst-case path. This means that large portions of the program are unavailable for instrumentation, because instrumenting them could affect the worst-case execution time and thus violate existing timing constraints. Based on this observation, we investigated program transformations to increase the effectiveness of time-aware instrumentation for such programs.

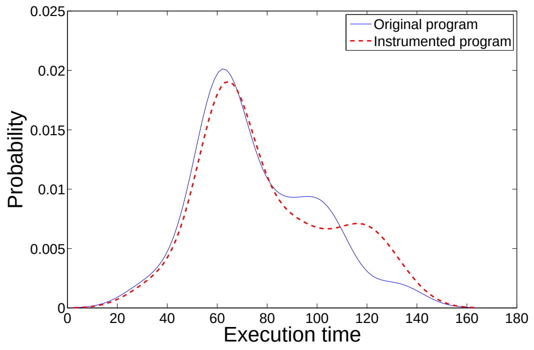


Fig. 1: ETPs of the OLPC keyboard controller [9]

This also raises the question of how to measure the effectiveness of a time-aware instrumentation approach, so that different approaches can be compared against each other. One way is a visual inspection as seen in Figure 1. This involves manual assessment and lacks accuracy. Another way is to calculate the instrumentation coverage as the ratio of extracted information to what is expected or desired. In this work, we propose the *ETP shift effectiveness* as a more efficient metric for time-aware instrumentation.

Finally, a central element for time-aware instrumentation is to identify regions in the program which can be instrumented. We propose an algorithm that identifies instrumentable edges in a program's control-flow. An instrumentable edge is one that lends itself for time-aware instrumentation. Different approaches can use these edges for program instrumentation. We demonstrate the utility of this algorithm by introducing Branch Block Creation and CFG Cloning as two such approaches to increase the effectiveness of time-aware instrumentation at the expense of code size. We measure the performance of these transformation approaches using the SNUbenchmark [10].

II. Time-aware Instrumentation Overview

Time-aware instrumentation aims to instrument a program while minimizing changes to the timing behavior on the worst-case path. Since the execution time of the program differs from one execution path to another, the idea is to instrument programs only in locations on non-worst-case paths. In the optimal case, this means zero added overhead to the program on the worst-case path.

The work flow for time-aware instrumentation differs from standard instrumentation in that it has extra steps to consider timing. Figure 2 shows the work flow of our approach. Initially, we analyze the program's source code, establish timing information, and generate its control-flow graph. The instrumentation tool (such as in previous work [9]) analyzes the program and considers timing information. The tool outputs an instrumentation configuration which the framework uses to compute an expected instrumentation coverage for a given set of variables. After running the tool, the developer checks whether the achieved expected coverage is acceptable. If the results are satisfactory, the developer will execute the instrumented program. Otherwise, if the expected coverage is insufficient, then the developer will have two means by which to attempt increasing the coverage. First, the developer can use our approaches to transform the program into a program that is more suitable for instrumentation. Second, the developer can change the debugging budget given to the instrumentation. This increases the number of instrumentation points. If neither of these two is successful, then the framework will report that it is unable to instrument and meet the desired coverage.

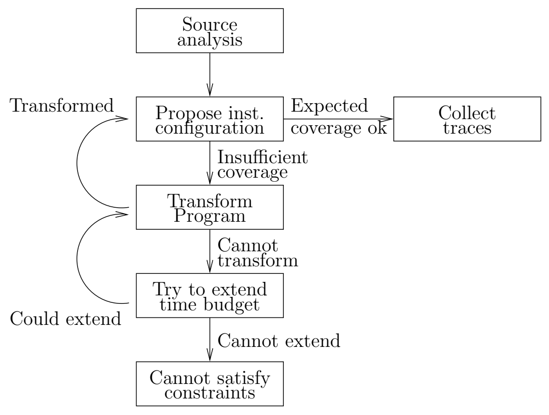


Fig. 2: Work flow for our tool chain.

The instrumentation process uses time differences between execution paths and different basic blocks to ensure that the instrumented program stays within the original program's time limits on the worst-case path. Sometimes, however, due to processor anomalies, cache behavior, etc., the timing might change. After each complete instrumentation attempt, the framework will analyze the worst-case execution time behavior to check if it exceeds the execution time of the original worst-case path plus the debugging budget. If this occurs, the framework will attempt a different instrumentation configuration which reduces the coverage on non worst-case paths to satisfy the timing requirements. This process will be iteratively repeated until the program meets its timing constraint.

III. Model and Terminology

Developers usually use instrumentation to trace information of interest such as the state changes of variables. To, for example, trace changes of a variable x, instrumentation must be applied to all places in the program that modify the variable x and add code to record its value. Time-aware instrumentation will choose which ones to instrument based on the timing constraints of the program.

The abstract model used to represent programs is similar to the one presented by Fischmeister and Lam [9]. A program is a directed graph $G = \langle V, E \rangle$ which captures the program's inter-procedural control flow. Each vertex $v \in V$ in the program's control-flow graph (CFG) represents a basic block in the program. A basic block is a unit of execution in the program and has a single entry and exit point. $E \subseteq V \times V$ is the set of edges that represent the flow of control in the CFG.

A path p_{v_s,v_d} of a CFG G describes a path from the source vertex v_s to the destination v_d with a sequence of vertices $\langle v_s, v_{s+1}, \ldots, v_{s+n-2}, v_d \rangle$. The worst-case path (WCP) is the path with the largest worst-case execution time (WCET) of all paths. We use measurement-based analysis to obtain the WCET and assume correct WCET measurements. The WCET analysis itself falls outside of the scope of this work, but it is in general technically feasible [11].

In addition to the differences between lower execution times and the WCET, instrumentation can also make use of the debugging time budget available to a program. A debugging time budget α is usually a small percentage of the program's slack. The program's slack is the time window between the program's deadline and WCET. For safety concerns, systems are designed with a slack that is sufficient to act as a safety assurance margin. The debugging budget α is a percentage of the CPU resources dedicated to debugging and must be accounted for in the schedulability analysis. Therefore, we can utilize the debugging budget for instrumentation in the manner described in Section II.

The instrumentation process might increase the WCET of a program by an overhead o. Perturbations in the WCET are acceptable as long as they are less than or equal to the debugging budget α, specified by the developer. Depending on the extent to which the system is loaded, the debugging budget may permit programs to absorb small increases in the WCET, and still ensure that the temporal deadlines are correct. So for single-task applications without interrupts, we must ensure that the overhead o is below the budget α. For instrumenting concurrent applications, we must ensure the schedulability of the whole workload after adding all overheads to the WCETs of the instrumented program functions. The specific way to distribute the available debugging budget among the tasks is up to the developer. However, a naive way to distribute the budget would be to assign weights to tasks (for instance following the tasks importance) and distribute the budget according to these weights.

IV. The ETP Shift Effectiveness Metric

Related work [9] used a coverage (also named reliability) criterion as a metric for the quality of instrumentation. The instrumentation coverage of an instrumented program is the ratio of the amount of information extracted at run time to the desired amount. So, for example, when the developer wants to trace 100 variable assignments and the instrumentation only yields 30 assignments, then the instrumentation coverage will be 0.3.

This metric fails to capture the potential for extracting information at an abstract level, because it only compares concrete solutions. For example, to compare two instrumentation techniques using instrumentation coverage, this metric measures the values for the different techniques for a particular execution of the program. Instead, a more useful metric can capture the quality of the different techniques over a wide range of inputs and, therefore, for different program executions. This metric should potentially estimate how much coverage each instrumentation technique achieves per unit time.

This work presents a new metric for time-aware instrumentation. The metric complements the previously explored coverage metric. The *ETP shift effectiveness* captures the potential for instrumentation and thus defines the optimal bound on time-aware instrumentation for any function based on its ETP. Figure 3a shows such ETPs for a fictive function. Values on the x-axis show the different execution times of the program and the y-axis shows the frequency at which the execution time occurs when executing this function.

Time-aware instrumentation bases on the idea of a *"right shift"* in the ETP during instrumentation. Figure 1 shows this right shift. The ETP of the instrumented program exhibits a right shift from the original program's ETP. The reason is that instrumentation utilizes paths with lower execution times (compared to the WCET). Instrumenting these paths increases their execution times and thus shifts the ETP to the right.

The *ETP shift effectiveness* uses this observation to quantify the theoretic optimum for time-aware instrumentation. The insight is that any software-based instrumentation inserts code in the programs and thus shifts the ETP. For a certain coverage, the less the shift in the ETP, the more effectively the method has used the slack (disregarding any execution time anomalies that might exist). This metric uses coverage/time as its basis with a double integral over time.

Figure 3 illustrates the *ETP shift effectiveness*. Figure 3a shows the ETPs of the original and different instrumented programs. Figure 3b shows the frequency distribution of the coverage over the execution time. Figure 3c shows the cumulative distribution function for Figure 3b. The more the shift in the ETP of the instrumented program, the further its cumulative distribution from that of the uninstrumented program.

Optimally an instrumentation technique would obtain full coverage without any shift to the program's ETP. The figures then show that as the effectiveness of the instrumentation increases, the cumulative distribution of the ETP gets closer to the optimum, i.e., to that of the uninstrumented program. Integrating the cumulative distribution curves once more obtains the effectiveness of the instrumentation. The smaller the value (and closer to that of the uninstrumented program), the more effective the instrumentation is.

The *ETP shift effectiveness* captures the achieved coverage per unit time of the instrumentation technique. So it is a measure of how successful an instrumentation technique is in consuming slack in the program (between lower execution times of non-worst-case paths and the WCET) for the sake of tracing a certain amount of information. For example, assume two instrumentation methods applied to the same program result in the same coverage but different *ETP shift effectiveness* values. The one with a lower value means a more efficient utilization of time to extract the same information.

V. Edge Detection for Program Transformation

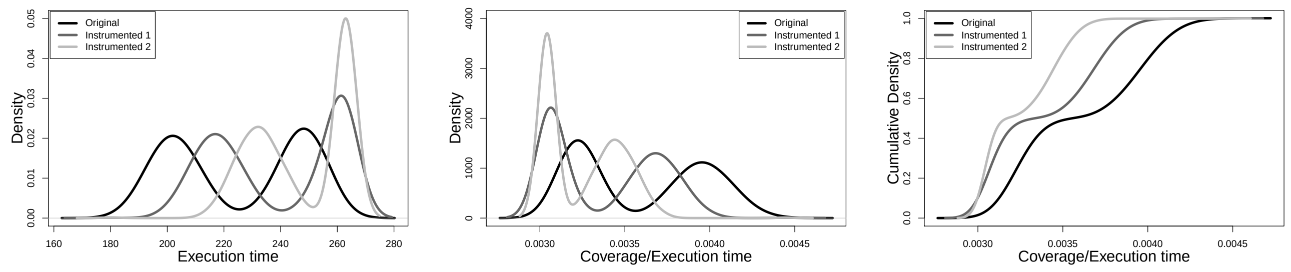
In a program's CFG, uninstrumentable non-WCP edges are ones that lie on non-WCPs and connect to basic blocks of the WCP. They are uninstrumentable because instrumenting any of the basic blocks to which they connect changes the WCET of the program. These uninstrumentable non-WCP edges limit the *ETP shift effectiveness* and are the basic elements used by methods that increase the coverage and the *ETP shift effectiveness*. Figure 4a shows a sample CFG with five basic blocks A, B, C, D and E. Assuming that $\langle A, B, C, D, E\rangle$ is the WCP, then the uninstrumentable non-WCP edge set includes $\langle A, C\rangle$ and $\langle C, E\rangle$, because these edges, although being non worst-case edges, they share all their basic blocks with the WCP.

We propose an algorithm to identify such edges as the basic building block for improving time-aware instrumentation. Function 1 shows this algorithm. The algorithm takes as input the CFG $G\langle V, E\rangle$ and the WCP p_{v_s,v_d} of the CFG, and it returns the set of edges of interest. The helper function *children* : $v \rightarrow V^{\{\}}$ returns the set of direct successor vertices of a vertex v. The queue operations *enqueue* and *dequeue* enqueue and dequeue an element from a queue, respectively.

This algorithm finds non-WCP edges that have subpaths of the WCP connecting their head and tail vertices. In Function 1, lines 5-6 iterate through all edges of the directed CFG G that have both vertices on the WCP p_{v_s,v_d}. For each edge e, line 7 queues the direct successors of the head vertex of the edge in the queue Q except for the tail of the edge e and line 8 marks the vertex as *visited*. Then, line 9 iterates on all vertices in the queue Q. Line 10 dequeues a vertex v from Q and lines 11-12 expand the direct successors of v and mark it as *visited*. Line 13 checks each of the direct successors of v, if the direct successor is the tail of the edge e, then line 14 will add edge e to set B. Otherwise, line 17 would enqueue the direct successor in Q, if it were not visited before. This algorithm is polynomial in time with respect to the number of vertices.

VI. Branch Block Creation

Branch Block Creation is a program transformation technique that uses the edge detection mechanism described in Section V. Branch Block Creation creates locations in the program for instrumentation. This increases the number of instrumentable basic blocks in the program.



(a) Sample ETPs (b) Coverage/ETPs (c) Cumulative distributions

Fig. 3: Illustrative example of the *ETP shift effectiveness*

Function 1 Edge Identification

```
Input: CFG $G$, $p_{v_s,v_d}$
Output: $E^{\{\}}$
 1: Let $S$ be the set of visited vertices
 2: Let $B \Leftarrow \emptyset$ be the set of edges for creating basic blocks
 3: Let $Q$ be an empty queue
 4:
 5: for $e = (v_a, v_b) \in E$ do
 6:     if $v_a \in p_{v_s,v_d}$ and $v_b \in p_{v_s,v_d}$ then
 7:         $Q \Leftarrow \mathit{children}(v_a) \setminus v_b$
 8:         $S \Leftarrow \{v_a\}$
 9:         while $Q \neq \emptyset$ do
10:             $v_x \Leftarrow \mathit{dequeue}(Q)$
11:             $S \Leftarrow S \cup \{v_x\}$
12:             for $v \in \mathit{children}(v_x)$ do
13:                 if $v = v_b$ then
14:                     $B \Leftarrow B \cup \{e\}$
15:                     break while loop
16:                 else if $v \notin S$ then
17:                     $\mathit{enqueue}(Q, v)$
18:                 end if
19:             end for
20:         end while
21:     end if
22: end for
23: return $B$
```

A. Overview

Figure 4 illustrates how Branch Block Creation transforms programs to increase the *ETP shift effectiveness* in time-aware instrumentation. Figure 4a shows a sample CFG with five basic blocks A, B, C, D, E. Assuming that $\langle A, B, C, D, E\rangle$ is the WCP, $\langle A, C, E\rangle$, $\langle A, C, D, E\rangle$, and $\langle A, B, C, E\rangle$ are non-WCPs that cannot be instrumented because they share all their basic blocks with the WCP. Applying Branch Block Creation modifies the CFG as shown in Figure 4b. We create two new basic blocks F and G on edges $\langle A, C\rangle$ and $\langle C, E\rangle$ (detected by Function 1), respectively. This creates new non-WCPs with basic blocks F and G that can be used for instrumentation given that the execution time of these paths stays less than or equal to that of the WCP $\langle A, B, C, D, E\rangle$.

The Branch Block Creation algorithm may modify the program's WCP, depending on the target architecture and compiler. After instrumentation, the end of basic block B (which used to be a fall-through block) now contains a new unconditional branch instruction to jump past the code at F. This instruction did not exist in the original program. Therefore, the creation of basic block F results in an overhead of one unconditional branch instruction on the WCP $\langle A, B, C, D, E\rangle$ (same happens due to G). Note that the location of the unconditional branch instruction, whether in the *if* or the *else* block, is architecture specific. If the instruction is in the *if* block then it will modify the WCP; otherwise it will not. Note also that even if the compiler adds the instruction to the *if* block, inverting the condition will move the instruction to the non-WCP (the instrumentation block) leading to an unmodified WCP. Therefore, modifying the WCP is avoidable. But since the avoidance is either architecture specific or requires code modification, this work assumes that the Branch Block Creation modifies the WCP.

B. Algorithm

The algorithm for Branch Block Creation iterates on the set of edges obtained as output from Function 1 and creates basic blocks on these edges. We use these created basic blocks for instrumentation either by instrumenting every basic block for modified variables or through the minimization of instrumentation points [9].

Branch Block Creation may add overhead on the WCP and this overhead must stay below the program's slack α (assuming no changes to avoid adding instructions on the WCP). This may lead to an increase in the WCET of the program. Thus, We want to choose only a subset of the basic blocks to create such that the overhead o is within the given budget α for instrumentation. Equation 1 describes this optimization problem:

$$\begin{aligned} &\text{Max} \sum_{i=1}^{n} b_i * (\mathit{vars}_i * \mathit{frequency}_i) \\ &\text{subject to} \sum_{i=1}^{n} b_i * (\mathit{overhead}_i * \mathit{frequency}_i) \leq \alpha \\ &\text{where } b_i \in \{0,1\} \text{ for } i = 1, 2, \ldots, n. \end{aligned} \quad (1)$$

vars_i is the number of traced variables at the created basic block i, $\mathit{frequency}_i$ is the number of times the basic block i executes (determined by the WCET analysis tool), and $\mathit{overhead}_i$ is the overhead on the WCP caused by creating the basic block i. n is the total number of created basic blocks from Function 1, and b_i is the binary variable.

Solving the problem of finding a subset of the basic blocks to create is NP-Complete. We can show this by first polynomially reducing the binary knapsack problem to this problem, thus proving that its NP-Hard, and then showing that the problem lies in NP (the reduction is omitted for space constraints). We solve the problem using binary integer programming (BIP).

The set of basic blocks to be created on the edges returned from Function 1 is given as input to this optimization problem. The output is the set of basic blocks to actually create and use for instrumentation. We use this subset of created basic blocks for instrumentation to satisfy the program's budget α.

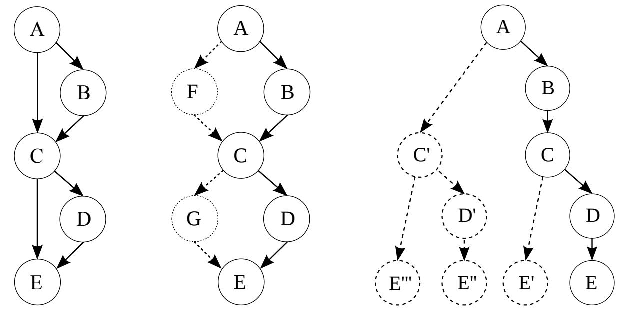
VII. CFG Cloning

In this section, we propose CFG Cloning as another transformation technique that uses the edge detection algorithm outlined in Section V. CFG Cloning facilitates instrumentation on non-WCPs that share basic blocks with the WCP. CFG Cloning does not add instructions to the WCP and offers more instrumentation flexibility at the expense of code size.

A. Overview

We illustrate the concept of CFG Cloning using the example CFG in Figure 4a. Again, we assume that $\langle A, B, C, D, E\rangle$ is the WCP. Although the CFG contains three other non-WCPs, we cannot instrument them, because they share all their basic blocks with the WCP.

CFG Cloning duplicates whole subgraphs of the CFG to permit instrumenting them. Figure 4c shows the CFG after we do CFG Cloning. First, for the edge $\langle A, C\rangle$, which does not fall on the WCP, we duplicate the basic block C and its subgraph. Edge $\langle C, E\rangle$ as well does not belong to the WCP and so we duplicate basic block E. It is worth noting that the edge $\langle C, E\rangle$ has been duplicated before (in the subgraph of C), and, therefore, we duplicate it twice, once for each occurrence. We choose to duplicate cloned occurrences because they represent different execution paths in the program and, hence, increase the number of locations at which instrumentation can be inserted. Now, each of the three non-WCPs that used to share basic blocks with the WCP, have their own paths with some unshared basic blocks.



(a) Original (b) BB Creation (c) CFG Cloning

Fig. 4: Example of program transformation

B. Algorithm

The algorithm for CFG Cloning iterates on the set of edges obtained as output from Function 1 and copies the tail basic blocks of these edges along with their subgraphs. The algorithm removes each of these edges and creates new edges from the head basic blocks to the copied subgraphs. This algorithm is polynomial in time with respect to the number of vertices in the CFG G.

One drawback of CFG Cloning is the potentially large increase in code size. In general, this increase is exponential, and Figure 4c already indicates this. This is because when a basic block is duplicated all its subgraph is duplicated as well. Moreover, we duplicate basic blocks as well as any copies of them that are created from the duplication of any ancestor basic block. Although we limit the CFG Cloning to the scope of functions and loops, the exponential duplication can still cause problems.

Given that there is a certain limit on the increase in code size, we want to choose only a subset of the basic blocks that the algorithm duplicates which maximizes the amount of traced information. We model this problem as a non-linear programming problem as shown in Equation 2.

$$\begin{aligned}
&\text{Max} \sum_{i=1}^{n} b_i * \mathit{vars}_i * \mathit{frequency}_i \\
&\text{subject to} \sum_{i=1}^{n} b_i * \mathit{code}_i \le \mathit{code}_{max} \\
&b_i = \prod_{j \in S_i} b_j \ \mathit{where}\ b_i \in \{0,1\} \text{ for } i = 1, 2, \ldots, n
\end{aligned} \quad (2)$$

Here, b_i is a binary variable designating a duplicated basic block i, S_i is the set of duplicated basic blocks upon which the existence of basic block i depends, vars_i is the set of traced variables at basic block i, $\mathit{frequency}_i$ is the number of times basic block i executes, and code_i is the amount of code added by duplicating basic block i. n is the total number of basic blocks available for duplication. For example, for the CFG in Figure 4c, we have three possible duplications C', E', and E''' (from E''). The code added by C' equals the total size of basic blocks C, D, E, while the code added by E', E''' is equal to the size of basic block E only. The existence of E''' would be valid only if C' was chosen for duplication.

After obtaining the new CFG G' from the CFG Cloning algorithm, the duplicated basic blocks are passed as input to the optimization problem in Equation 2. The output is only a subset of all duplicate basic blocks. The set of duplicate blocks that have not been selected will be removed from the CFG G' and their subgraphs. The edges that were removed for creating these vertices will have to be reconstructed. This can easily be done because the edges are already stored in the set B in Function 1.

VIII. Experimentation

We explore the two transformation methods in practice using the SNU real-time benchmark suite [10]. This benchmark suite contains 17 C programs that implement numeric and DSP algorithms. The benchmarks have on average 117 lines of code and 34 basic blocks. We extended the benchmarks with a wide range of inputs to generate reasonable ETPs.

We apply the program transformation techniques to all benchmarks before instrumenting them. For the actual instrumentation, we use the technique proposed by Fischmeister et al. [9]. We compare the instrumentation of the transformed benchmarks against the instrumentation without transformation. Note that the transformation and instrumentation process is fully automated. We use CIL [12] for static code analysis and control flow graph extraction.

All experiments were run on a Keil MCB1700 board running a 100 MHz ARM Cortex-M3 processor-based MCU. Trace data from each benchmark was logged in a buffer and sent off-chip to a PC monitor for analysis. Note that a task sends data off chip at the end of a super loop as in a cyclic executive system. We use RapiTime [13] to analyze the WCET of the programs. We trace all variable assignments except for function arguments, constants, and loop counters. We set a debugging budget α for each program that is 2% of its WCET.

The goal of experimentation is to quantitatively assess the transformation techniques using the following metrics:

- ***ETP shift effectiveness*:** This metric indicates the extent by which an instrumentation method utilizes time for instrumentation coverage.
- **Average instrumentation coverage:** Instrumentation coverage shows the effectiveness of an instrumentation method in capturing variable assignments. For each benchmark, we calculate the coverage for every input and compute the average across all inputs.
- **Instrumentation time:** This metric shows the time the tool spends in parsing the code, instrumentation, optimizing for debugging budget α, and any retries required. The instrumentation tool will retry the instrumentation with reduced overhead, if the WCET overhead after instrumentation exceeds the debugging budget α.
- **Increase in code size:** Every instrumentation point adds extra code to the program. The less the increase in code size, the more effective the instrumentation approach is in utilizing code space for instrumentation.
- **Number of retries:** It shows how often the instrumentation tool reduces the instrumentation coverage due to exceeding the debugging budget α. A small number of retries is essential for the applicability of the approach.

A. Results

Figure 6 shows the average instrumentation coverage for the different instrumentation approaches. The error bars show the maximum and minimum coverage over all executions of each benchmark. For the benchmarks: *fft1k*, *fibcall*, *insertsort*, *jfdctint*, and *matmul*, none of the instrumentation approaches was able to extract any information. This happens, because either the program has a single path which is the WCP and cannot be instrumented, or it has multiple paths but adding any instrumentation code modifies the program's WCP.

Table I shows the results for the *ETP shift effectiveness*, instrumentation time, increase in code size, and number of retries for each of the instrumentation approaches. The values of the *ETP shift effectiveness* are normalized to those of previous work instrumentation. We omit the data of the five benchmarks for which none of the instrumentation approaches was able to extract any traces. The instrumentation for all benchmarks is limited by a 2% debugging budget α, and if exceeded, the instrumentation tool will repeat the instrumentation while reducing coverage. The benchmarks *qsort-exam* and *select* show the increase in code size when using CFG Cloning.

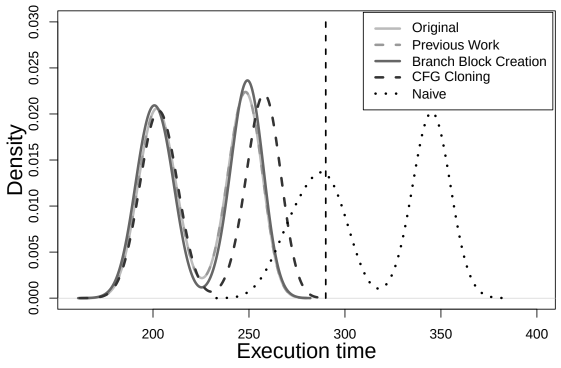


Fig. 5: Execution time profiles for *qsort-exam*

Figure 5 illustrates the benefit of the proposed instrumentation approaches. The vertical line represents the WCET plus a 2% debugging budget α. The figure shows the original ETP of the *qsort-exam* benchmark. It also shows the shifted ETPs with four different instrumentation approaches; previous work, Branch Block Creation, CFG Cloning, and naive instrumentation. The time-aware instrumentation techniques shift the original within the debugging budget α. Naive instrumentation instruments for variables of interest without taking timing into account. Although, the instrumented program achieves full coverage, its ETP fails to obey the timing requirement.

B. Discussion

In 11 out of 12 instrumentable benchmarks, Branch Block Creation and CFG Cloning always increase the instrumentation coverage. For the benchmark bs, all instrumentation methods perform alike. In many cases, both transformation techniques perform equally well, in terms of coverage, except for $select$ in which CFG Cloning performs better. The reason is that Branch Block Creation has less basic blocks to instrument on the execution path, whereas CFG Cloning clones basic blocks along with their subgraphs thus more instrumentable basic blocks. Hence, using CFG Cloning adds more flexibility to the instrumentation process leading to an increase in the instrumentation coverage for some executions of the programs.

ETP shift effectiveness is effective in identifying efficient instrumentation methods. The better the utilization of slack on the non-WCPs for the obtained coverage, we get a smaller value of *ETP shift effectiveness*. For the *adpcm-test* benchmark, for example, Branch Block Creation and CFG Cloning increase the coverage from 0.003 to 0.03 compared to previous work. Their *ETP shift effectiveness*, however, is almost 10 times that of previous work. In such case, it is up to the developer to decide whether such a shift in the program's ETP is acceptable for the corresponding increase in coverage. In

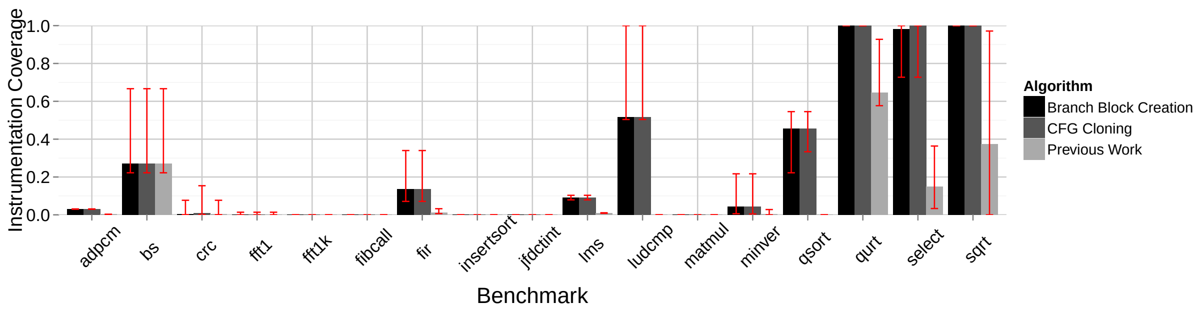


Fig. 6: Average instrumentation coverage with the range showing as the error bars

some cases, the coverage obtained by Branch Block Creation and CFG Cloning are equal but the values obtained from *ETP shift effectiveness* do not match such as for benchmark *qsort-exam*. The reason is that although both essentially extract the same amount of information, the CFG Cloning utilizes less slack for instrumentation. *ETP shift effectiveness* is thus useful for choosing an instrumentation metric over the other if, for example, their coverage match.

The instrumentation time of the benchmarks is acceptable and has a maximum of 420 mS. As a test for scalability, we concatenated all benchmarks into one C file of about 3200 lines of code. The instrumentation time was 653 mS. In most cases, the first instrumentation attempt was successful in honoring the debugging budget α. The tool had to adjust the instrumentation in some cases with a maximum of 4 retries.

Out of 17 benchmarks, 5 are not instrumentable even after transformation. Two benchmarks are instrumentable only after transformation. Branch Block Creation and CFG Cloning increase the instrumentation coverage, on average over all benchmarks, compared to previous work by 5.8 and 5.9 times, respectively. CFG Cloning increases the program code size considerably but gives more flexibility to instrumentation and has a better utilization of slack in most of the cases. The *ETP shift effectiveness* is an indicator of the effectiveness of the instrumentation approaches.

IX. Discussion

This section focuses on some high-level issues regarding the applicability of the results and the proposed techniques.

Code size and memory profile: The program transformation techniques proposed in this work can modify the block layout of the program especially the CFG Cloning. Modifying the layout can change memory and cache profiles, which in turn may change the WCET of the program. After each complete transformation attempt, the framework analyzes WCET measurements to check for such changes. If this happens and the budget α disallows the change, the framework will attempt a different transformation configuration. For example, the framework will reduce the number of cloned or inserted blocks. This process repeats until the requirements are met.

Scope and traceability: Although the examples and experimentation in this work focus on tracing data variables, time-aware instrumentation can similarly trace function calls, or control flow. The main goal of time-aware instrumentation is information extraction without affecting a program's worst-case timing constraint. This limits the amount of extracted information due to instrumenting only subsets of the basic blocks and avoiding instrumentation of the WCP. Partial traces, however, are useful for program understanding, performance analysis, and optimizations [14], [15].

Multiple WCPs: In our analysis, we ignored the exceedingly rare case of multiple WCPs in a program. This case never occurred in the experiments. However, addressing multiple WCPs is an easy task. Our instrumentation tool would simply avoid instrumenting any WCP. The algorithm for finding instrumentable edges will then take multiple WCPs as an input.

WCET analysis tools: In our experimentation, we used RapiTime [13] to obtain the WCET of the basic blocks. RapiTime is a measurement-based WCET analysis tool and thus might underestimate the actual WCET. The WCET, however, is only an input to the instrumentation tool and thus the validity of the proposed concept is independent of the accuracy of the analysis tool. The choice of RapiTime as a WCET analysis was due to the availability of the tool in our labs, past experience using it, and independence of the architecture on which the software executes. It is also the de facto industrial standard applied in fields like aerospace and automotives. We can obviously replace RapiTime with a static analysis tool such aiT [16] to obtain WCETs, but this is also known to be costly for modern architectures.

Optimization criteria: In Sections VI and VII, the algorithms only focus on optimizing for the budget α and code size, respectively. The value of a basic block considers only the number of traced variables and the frequency of executing the basic block. Other criteria can be considered such as the usefulness of the traced variables from a tracing perspective that allows for choosing more optimal basic blocks.

X. Related Work

Some instrumentation tools are capable of inserting instrumentation points to binary executables. Binary static instrumentation tools include Etch [17] the program performance evaluation and optimization system, Morph [18], QPT [19], EEL, and ATOM. Examples of dynamic instrumentation tools that do code transformation during program execution are Pin [4] and DynamoRIO [5]. Other examples of dynamic binary instrumentation also include DTrace, SystemTAP, Frysk and GDB. These tools overwrite code locations with trap

TABLE I: The *ETP shift effectiveness*, the overhead on the WCP, and the increase in code size for different approaches

Benchmark	***ETP shift effectiveness***			**Instrumentation Time [mS]**			**Increase in code size [bytes]**			**Retries**		
	Previous	**Creation**	**Cloning**	**Previous**	**Creation**	**Cloning**	**Previous**	**Creation**	**Cloning**	**Previous**	**Creation**	**Cloning**
adpcm-test	1	10.191	9.968	167	177	420	8	84	468	0	0	4
bs	1	1	1	30	38	48	16	16	16	1	1	1
crc	1	0.989	1.869	32	44	61	20	28	608	1	1	1
fft1	1	1	1.009	57	69	79	28	44	88	0	0	0
fir	1	9.813	9.813	108	91	91	36	48	52	1	0	0
lms	1	8.933	8.933	59	66	70	32	40	44	0	0	0
ludcmp	1	0.0039	0.0039	62	67	70	0	20	108	0	0	0
minver	1	8.146	7.643	100	104	109	24	144	512	0	0	0
qsort-exam	1	0.043	0.023	55	63	104	31	40	1628	0	0	2
qurt	1	0.915	0.915	32	41	51	24	32	36	0	0	0
select	1	6.120	3.995	69	101	135	20	144	2,000	0	1	3
sqrt	1	0.889	0.889	29	33	35	24	32	36	0	0	0

instructions to execute instrumentation code.

Mellor-Crummey et al. propose a software instruction counter [1] to debug parallel programs using the integrated approach to parallel program debugging on large-scale shared-memory multiprocessors introduced by Fowler et al. [20]. Thane [3] and Dodd et al. [2] present integrated approaches for monitoring and debugging of real-time systems. Moore et al. [6] and Omre [7] introduce hardware trace debuggers.

All these instrumentation methods are known to affect the behavior of the program including its temporal behavior which is sometimes not acceptable in real-time embedded systems. One method for preserving timing constraints is partial instrumentation [14], [15]. Another method is doing time-aware instrumentation to only instrument at locations that do not affect the timing behavior of the program [9].

XI. Conclusion

Instrumentation for information extraction supports understanding specific aspects and behavior of the software at run time. Time-aware instrumentation tries to preserve logical correctness *and* timing constraints during instrumentation.

This work introduces the *ETP shift effectiveness* as a new metric for measuring the performance of time-aware instrumentation techniques. We discuss two approaches using the metric and measure their effectiveness. While the two approaches are straightforward, they and the new metric lay the foundation for future work for more complicated approaches as well as for instrumentation mechanisms going beyond timing and logical correctness.

XII. Acknowledgments

This research was supported in part by NSERC DG 357121-2008, ORF-RE03-045, ORF-RE04-036, ORF-RE04-039, APCPJ 386797-09, CFI 20314 and CMC, ISOP IS09-06-037, and the industrial partners associated with these projects.

References

[1] J. M. Mellor-Crummey and T. J. LeBlanc, "A Software Instruction Counter," in *Proc. of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*. New York, NY, USA: ACM, 1989.

[2] 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.

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

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

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

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

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

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

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

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

[11] 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, 2008.

[12] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *In International Conference on Compiler Construction*, 2002, pp. 213–228.

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

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

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

[16] C. Ferdinand and R. Heckmann, "ait: Worst case execution time prediction by static program analysis," *Ifip International Federation For Information Processing*, vol. Volume 156, pp. 377–383, 2004.

[17] 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 *Proc. of the USENIX Windows NT Workshop*. Berkeley, CA, USA: USENIX Association, 1997.

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

[19] J. Larus, "Efficient program tracing," *Computer*, vol. 26, no. 5, 1993.

[20] R. J. Fowler, T. J. LeBlanc, and J. M. Mellor-Crummey, "An integrated approach to parallel program debugging and performance analysis onlarge-scale multiprocessors," *SIGPLAN Not.*, vol. 24, 1988.