Statistically Analyzing Execution Variance for Soft Real-Time Applications

Tushar Kumar[1], Romain Cledat[2], Jaswanth Sreeram[2], and Santosh Pande[2]

[1] School of Electrical and Computer Engineering,
Georgia Institute of Technology
`tushark@ece.gatech.edu`
[2] College of Computing,
Georgia Institute of Technology
`romain@cc.gatech.edu, jaswanth@cc.gatech.edu, santosh@cc.gatech.edu`

Abstract. Certain high-performance applications like multimedia and gaming have performance requirements beyond reducing program execution time. These applications have repetitive components whose desired performance characteristics are more naturally expressed using soft real-time theory with its probabilistic guarantees. However, for large complex gaming and multimedia applications, programmers typically avoid real-time constructs as they significantly constrain how the programmer can express functionality. Instead, such applications are developed as monolithic programs using conventional languages like C/C++. Here the soft real-time behavior of the application becomes an emergent quality rather than being enforced by design. Programmers must then tweak parameters/algorithms until the application's soft real-time behavior becomes acceptable. There are currently no analysis techniques that directly extract the soft real-time execution characteristics of monolithic applications written without the use of real-time constructs. We introduce a domain-agnostic profiling methodology that identifies program execution-contexts whose variant behavior most significantly affects the soft real-time characteristics of the application.

1 Introduction

Important classes of high-performance applications, such as gaming and multimedia have performance requirements beyond minimizing program execution time. These applications have Quality-of-Service (QoS) requirements on repetitive application components, such as a live-video encoding application that attempts to maximally compress the input image stream while maintaining a sufficiently smooth frame-rate. Other examples include real-time object tracking and recognition kernels at the heart of many military and commercial applications.

Typically, games, streaming live-video encoders and video players attempt to maintain a reasonably high frame-rate for a smooth user experience. However, they frequently drop the frame-rate by a small amount and occasionally by a large amount if the computation requirements suddenly peak or compute resources get taken away. Therefore, the QoS requirements of such applications is best described using a combination of *i)* soft real-time theory with its probabilistic guarantees, and *ii)* the runtime sophistication of certain application-specific artifacts, such as the degree of compression achieved on a raw video stream, or the realism of Artificial Intelligence or physics modeling in games. During the design optimization stage, programmers like to tweak algorithms and parameter values in order to maximize the runtime sophistication of their application while minimally compromising the desired real-time characteristics. *In this paper, we exclusively focus on characterizing the soft real-time behavior of an application, in the absence of any knowledge about the application's functionality or its domain.* We limit our technique to informing the programmer about an application's soft real-time behavior, and leave it to the programmer to decide how best to tweak algorithms based on application and domain specific knowledge.

Monolithic Applications Programmers often use specialized real-time languages and libraries to either guarantee that real-time requirements are met (hard real-time, for safety critical applications), or are met as close as possible (soft real-time). Such programming requires that the application be broken up into real-time constructs such as tasks, ordering dependencies be established between tasks, and completion deadlines (probabilistic or hard) be set on the tasks. However, when developing large applications like gaming and video, programmers typically eschew the benefits of formal real-time methods and languages, instead using conventional C/C++ development flows for their significant high-productivity advantages. The soft real-time behavior of the resulting monolithic application becomes entirely an *emergent quality* rather than being enforced by design. Programmers are then left to use ad-hoc means to understand what application components are responsible for undesirable soft real-time behavior.

In order to rectify the lack of suitable analysis tools for such applications, we propose a profile-driven methodology for characterizing the soft real-time behavior of *conventionally written monolithic* applications. The primary objective of our profiling methodology is the identification of application components that exhibit the *maximum variability* in their execution time. Such components are the ones most likely to affect the meeting of *implied* execution deadlines (such as desired frame-rates).

Application Structure A soft real-time application typically processes a sequence of data items, such as a sequence of image-frames for MPEG video encoding. There are soft real-time requirements limiting the average execution time and variability in execution time for functions that process the data sequence. A programmer unfamiliar with the application stands to gain important insights about the application's design and functionality if the most significant functions processing the data sequence are pointed out. Our profile analysis framework automatically identifies those functions whose repetitious behavior most significantly contributes variance to their enclosing scopes. Consequently, the set of functions identified by the profile analysis can be expected to closely match the set of functions that process the data sequence. The primary intuition behind this reasoning is as follows: the application needs soft real-time requirements to be enforced primarily because processing each data item, or parts of a data item, does not take constant time. Isovic, et al. [1] show that there is a significant amount of variation in decoding times for realistic video streams and argue that standard scheduling algorithms that assume average values and limited variation in frame decoding times will lead to poor video quality.

More generally, a soft real-time application may exhibit variability at many levels: from the highest-level of processing a data-item, to lower-levels of processing pieces of the data-item. Examples of this are image-frames at the highest-level in video-encoding, and motion-estimation over 8-pixel $\times$ 8-pixel blocks of the image-frame. The execution-time of motion-estimation may vary dramatically from block-to-block even within the same image-frame, depending on how wide the motion-estimation searches to find a closely matching block. There is a large body of research showing how the search-space of motion-estimation can be limited based on the types of input video expected or the search-space dynamically adapted in order to more consistently achieve the desired frame-rate. Our profiling framework helps the programmer identify functions contributing significant variability at all levels of processing in the application, and empowers the programmer to make decisions about whether, where and to what extent algorithmic or configuration-parameter tweaking needs to be done, such as adjusting parameters that constrain the size of the motion-estimation search window.

1.1 Research questions

We posed the following open research questions in order to drive the design of our profile analysis framework:

Question 1. **Component discovery** Can recurrent behavior identified during profiling of function calls be used to identify individual components of an application's soft real-time functionality?

Question 2. **Structure discovery** Can the identified recurrent behavior be used to reconstruct the soft real-time structure of the application? The structure would be composed of components of soft real-time functionality.

Question 3. **Context-sensitivity discovery** Can the context-dependent variability in the behavior of soft real-time components be detected? The behavior of a component may differ significantly depending on where it is invoked.

Question 4. **Generality** How reliably can behavior discovered during profiling be expected to generalize to future runs of the application on arbitrary inputs?

1.2 Contributions

We make the following specific contributions in this paper.

– We describe a tractable approach for succinctly capturing the behavior of millions of profile events in terms of tens of soft real-time components. The discovered components are functions that introduce significant variability to the application's real-time behavior, and hence are most important to be brought to the attention of a programmer interested in improving soft real-time behavior.

– We demonstrate that function call-chain segments capture the context-sensitivity of a component's soft real-time behavior. We motivate how the length of call-chain segments gives them varying ability to differentiate between multiple contexts of execution of a component. We provide algorithms that choose the correct segment lengths in order to produce highly succinct profile results that differentiate only between those contexts where behavior is significantly different.
– We illustrate the use of specific statistical theory for constructing algorithms that find *patterns* of behavior (dominant components and corresponding execution-contexts). Due to probabilistic guarantees provided by the statistical theory, the produced patterns generalize well for describing the behavior of the application executing on arbitrary input data.

We validate the correctness of the identified components by profiling well-known multimedia applications. Extensive prior research exists about the soft real-time behavior of these applications. The components reported by our profiling methodology match closely with those described in prior research as the main causes of soft real-time variance in these applications.

Among the questions listed above, only the structure discovery question is not satisfactorily answered by our current methodology. Although the discovered components and their call-chain contexts do allow the programmer to infer the structure, this inference is not sufficiently precise and may not work in all circumstances. In section 6 we provide insights on how our technique can be improved to accomodate structure discovery as well.

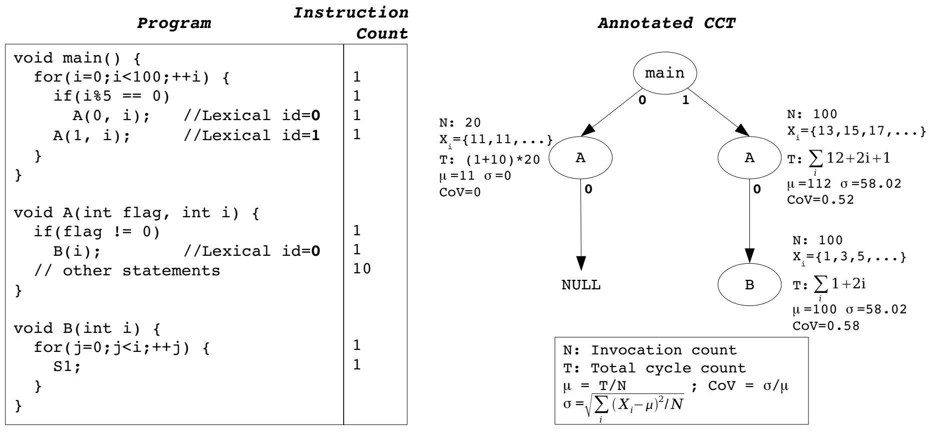
Overview Section 2 introduces the profile representation constructed from the raw stream of profile events. Section 3 introduces the relevant statistical theory and describes the analysis performed on the constructed profile representation for detecting patterns. Section 4 provides experimental validation.

2 Profile Representation

We profile-instrument the application and use the generated profile events to construct a Calling Context Tree (`CCT`). Ammons, et al. [2] showed that a `CCT` representation succinctly captures the dynamic structure of the function calls executed by the application. It preserves the full call-chain context of invocation and merges information along multiple identical paths into a single path. This makes the `CCT` an ideal representation for investigating context-sensitive behavior.

We automatically profile-instrument a C/C++ application using the LLVM [3] compiler infrastructure. We execute the application on test inputs and use the generated sequence of profile-events to construct the `CCT` as described in [2].

During CCT construction, we annotate statistics on each CCT node. These statistics are used by subsequent analysis for detecting patterns. Figure 1 shows an example program, the corresponding CCT and annotated node statistics. Function A was invoked from two call-sites within the parent function main. This leads to two children nodes for function A. Since function B was never invoked under the left A node, it only gets a *NULL* edge at its call-site in A. The next subsection describes the node annotations required for our variant call-context analysis.



```
Program                               Instruction
                                            Count
void main() {
  for(i=0;i<100;++i) {                 1
    if(i%5 == 0)                       1
      A(0, i);    //Lexical id=0       1
    A(1, i);      //Lexical id=1       1
  }
}

void A(int flag, int i) {
  if(flag != 0)                        1
    B(i);         //Lexical id=0       1
  // other statements                  10
}

void B(int i) {
  for(j=0;j<i;++j) {                   1
    S1;                                1
  }
}
```

Fig. 1. Sample Program and CCT with Annotated Node Statistics

2.1 Node Annotations

Each node is annotated with the following statistics about the execution of the function-call corresponding to it:

1. invocation count N: The number of times the corresponding function-call was invoked.
2. mean $\bar{X}$: The mean execution time across all invocations of the function-call corresponding to the node. *This includes the execution time of all children function-calls.*
3. variance σ^2: The statistical variance in the execution time of the function-call across all invocations. Variance is the square of the standard deviation σ. Using $\sigma^2 = E(X^2) - \bar{X}^2$, a single pass over the profile data constructs the CCT and computes all node annotations including variance.

2.2 Measuring Execution Time

We need to use some notion of elapsed time to time-stamp each function entry and exit event. Ideally, we would like to use wall-clock time with the application running on the target platform. Initially, we need to profile-instrument each function entry and exit since we don't know which ones will be significant for determining an application's variant behavior. However, this approach will suffer such a large runtime overhead that wall-clock measurements will be rendered meaningless for capturing the application's real-time behavior.

In order to avoid introducing significant distortions to the application's real-time behavior, we chose to use dynamic-instruction-counts to estimate elapsed time. While measuring dynamic-instruction-count does not account for micro-architectural effects and system level stalls (such as cache misses), it does allow us to robustly compare execution times in the *order-of-magnitude* sense for function-call instances in the `CCT`. Our primary motivation is to determine which function call instances (`CCT` nodes) are highly variant with respect to their mean execution times, and which function call instances are orders-of-magnitude more variant than others. Time-stamping profile events with dynamic-instruction-counts suffices for this purpose, while at the same time remaining unaffected by the overheads of profile-instrumentation. The LLVM compiler pass inserts code to update a global counter for the dynamic-instuction-count. This counter is sampled when dumping profile events during program execution.

Once the pattern generation analysis has been performed using dynamic-instruction-counts to measure elapsed time, the function names that *cumulatively* (i.e., over all `CCT` instances of same function) consume significant execution time are known. In subsequent iterations of profiling, real wall-clock time can be used to dump profile events only for those functions that were previously seen to consume significant time cumulatively. This would dramatically lower the runtime overhead of profile-instrumentation since lower-level functions that do not affect the overall analysis would not get profile-instrumented at all. The resulting wall-clock measurements can then be expected to closely match the real-world execution timing of the application.

3 Detecting Patterns of Behavior

Once the `CCT` has been constructed and its node annotations calculated, the `CCT` is traversed in pre-order for analysis. Nodes whose total execution time constitutes a miniscule fraction (say, $< 0.02\%$) of the total execution time of the program and their children subtrees, are deemed as *insignificant.* All other nodes are deemed *significant.* Since `CCT` nodes subsume the execution time of their children nodes, once a node is found to be insignificant, the nodes in its children subtree are guaranteed to be insignificant as well.

Since insignificant nodes individually constitute a miniscule portion of the program's execution time, any patterns of behavior detected for them would quite likely provide very limited benefits in optimizing the design of the whole application. Therefore insignificant nodes are ignored from all further analysis. This dramatically reduces the part of the `CCT` that needs to be examined by any subsequent analysis, leading to considerable savings in analysis time. For each application, the programmer can experimentally tweak the cutoff percentage used to determine significant nodes. A good methodology for this would be to start with a relatively large setting (say, $< 0.1\%$) and successively reduce it until the profiling results stabilize. Stabilization suggests that further inclusion of less significant nodes does not affect the analysis.

3.1 Tagging Nodes

We examine the annotations of nodes to determine if the corresponding nodes exhibit high-variance in execution-time within the context of the caller function (parent node). This is captured by the variance $P.\sigma^2$. We use Chebyshev's inequality [6] given below to determine meaningful thresholds to compare a node's variance. **Chebyshev's inequality** establishes conservative probability bounds on a given collection of data samples *while making no assumptions about the underlying probability distribution that generated the data.*

$$Pr(|X - \mu| \geq k\sigma) \leq \frac{1}{k^2} \tag{1}$$

In our experiments, we define a node to be high-variant if its execution time cannot be guaranteed to lie within 200% of its mean with atleast 96% probability. This implies $\frac{1}{k^2} = 1 - 0.96 = 0.04$ and $k\sigma = 2 \times \mu$. Therefore $\frac{\sigma}{\mu} \geq 0.4$ becomes the condition for high-variance. Consequently we use the Coefficient-of-Variability metric for classifying the variant-nature of nodes: $CoV = \frac{\sigma}{\mu}$. The choice of the variance-window around the mean and the probability of samples falling within it can be tweaked by the user based on the method described above. As the programmer pushes the probability guarantee of samples falling within the $k\sigma$ variance window to 100%, $\frac{1}{k^2} \rightarrow 0$ and $k \rightarrow \infty$. This implies that the window $k\sigma \rightarrow \infty$ would trivially encompass all possible execution-times. Therefore, it is practical to keep the probability not too close to 100%, and for almost all soft-real-time applications a probability guarantee of 96% would suffice, though this can be adjusted to the guarantees desired for any given application. Qualitatively, $k\sigma = 200\% \times \mu$ suggests a highly variant behavior as the execution-time can increase to over thrice the mean execution time (and reduce all the way down to 0). Based on how stringent the soft real-time requirements are for an application, the programmer can adjust the threshold that defines high variant behavior.

Once the `CCT` is constructed from the profile data, it is pre-order traversed in linear-time and individual nodes may be *tagged* as being *high-variant*. As mentioned earlier, the traversal is restricted to significant nodes.

3.2 Signature Generation for Patterns

The previous subsection described how significant nodes in the `CCT` were individually tagged if they exhibited statistical high-variance. The next step is to find *patterns of call-chains* whose presence on the call-stack can be used to predict the occurence of the high-variance behavior found at the tagged nodes. For a given tagged node P, we restrict the call-chain pattern to be some contiguous segment of the call-chain that starts at `main` (the `CCT` root node) and ends at P.

The names of the sequence of function-calls in the call-chain segment become the detection pattern arising from the tagged node. This particular detection pattern might occur at other places in the significant part of the `CCT`. Quite possibly, the occurence of this detection pattern elsewhere in the `CCT` does not match the statistical behavior, i.e., mean and CoV values, that were observed at the tagged node. Therefore, our key criteria in generating the detection pattern is the following:

> All occurences in the significant `CCT` of a detection pattern arising from a high-variance tagged node must exhibit the same statistical behavior that was observed at the tagged node.

Notice that this condition is trivially satisfied if we allow our detection pattern to extend all the way to `main` from the tagged node, since this pattern cannot occur anywhere else due to its full call-context being a unique path in the `CCT`. In many applications, patterns extending to `main` are likely to *generalize* very poorly to the *regression execution* of the application on arbitrary input data. Regression execution refers to the real-world-deployed execution of the application, as opposed to the *profile execution* of the application that produced the profile sequence used for constructing the `CCT`. In many applications, we expect the behavior of the function call at the top of the stack to be correlated with only the function-calls just below it in the call-stack. This short call-sequence would be expected to produce the same statistical behavior regardless of where it was invoked from within the program (i.e., regardless of what sits below it in the call-stack). In this paper we focus our attention on detecting just such call-sequences. We call these *Minimal Distinguishing Call Sequences* (`MDC` sequences) corresponding to any particular statistical behavior. These are the shortest length detection sequences whose occurence predicts the behavior at the tagged node, with no false positive or false negative predictions in the `CCT`. A pattern with `MDC` is illustrated in Figure 2.

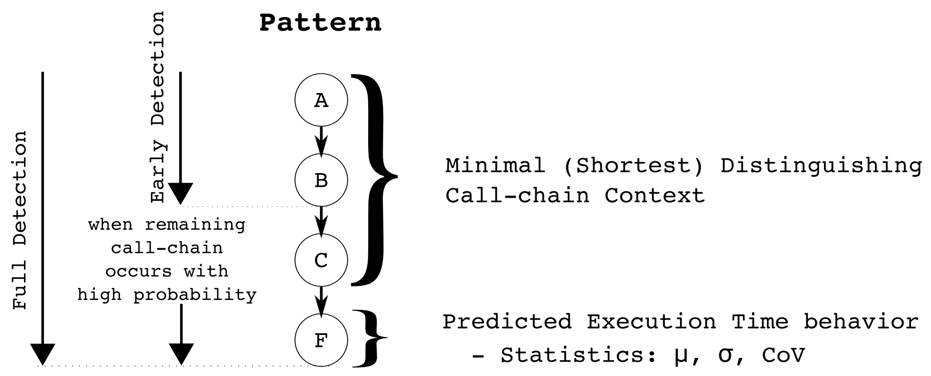

Pattern
Full Detection
Early Detection
when remaining call-chain occurs with high probability
A
B
C
F
Minimal (Shortest) Distinguishing Call-chain Context
Predicted Execution Time behavior - Statistics: μ, σ, CoV

Fig. 2. Minimal Distinguishing Call-Context Pattern

Given a tagged node P, Algorithm 1 produces the `MDC` sequence for P that is just long enough to distinguish the occurence of P from the occurence of any other significant node that has the same function-name as P but does not match the statistical behavior of P (the *other_set*). This is done by starting the `MDC` sequence with a call-chain consisting of just P, and then adding successive parent nodes of P to the call-chain until the `MDC` sequence becomes different from every one of the same length call-chains originating from nodes in the *other_set*. Therefore, by construction, using steps 6 - 9 of Algorithm 1, the `MDC` sequence cannot occur at any `CCT` nodes that do not satisfy the statistics of P (matching mean and CoV). However, the same `MDC` sequence may still occur at multiple nodes in the `CCT` that *do* satisfy the statistics for P (at some nodes in the *match_set* in step 5). There is no need for P's `MDC` sequence to distinguish against these nodes as they all have the same statistics and correspond to the call of the same function as for P. Since all nodes in the *match_set* will have the same *other_set*, the algorithm is optimized to generate the *other_set* only once, and apply it for all nodes in the *match_set* even though only P was passed as input. The algorithm outputs the `MDC` sequence for each node in *match_set* (called the *Distinguishing Context* for P).

Algorithm 1: Minimal Distinguishing Call Sequence Generation

```
Input: CCT $C$, Tagged CCT Node $P$
Output: Distinguishing Context $DC$ for $P$: set of pairs of form <MDC sequence,
        node of occurence>
1  begin
2      $DC \longleftarrow \emptyset$;
3      $func\_name \longleftarrow$ Name of function corresponding to node $P$;
4      $all\_set \longleftarrow$
       $get\_all\_significant\_node\_occurences\_of\_function\_in\_CCT(func\_name, C)$;
5      $match\_set \longleftarrow identify\_all\_nodes\_with\_matching\_statistics(P, all\_set)$;
6      $other\_set \longleftarrow all\_set - match\_set$ ; // identify nodes with same name
       whose statistics don't match $P$'s
7      for each CCT node $m$ in $match\_set$ do
8          MDC $\longleftarrow [< func\_name,$ lexical-id of $m$ in its parent $>]$ ; // initialize
           MDC as call-chain of length 1
9          Extend MDC sequence with parent nodes of $m$ (and their lexical-ids) until
           the detection pattern MDC is different from call-chains of same length
           arising from every node in $other\_set$;
10         $DC \longleftarrow DC \cup \{< \texttt{MDC}, m >\}$;
11 end
```

3.3 Grouping and Distinguishing between Similar Patterns

In the previous discussion, we were assuming that the programmer desired to distinguish between tagged nodes whenever their statistics (mean, CoV) didn't match exactly. However, exact matching of statistics may lead to very long detection patterns that generalize poorly to regression runs. For example, if multiple high-variant tagged nodes with very different means require long call-chains to distinguish between each of them, then it may be preferable to actually have a shorter call-chain pattern that does not distinguish between the tagged nodes.

Furthermore, if the same detection sequence occurs at multiple tagged nodes in the significant CCT and the nodes have matching statistics, we would like to combine the multiple occurences of the detection sequence into a single detection sequence. Such detection sequences are likely to generalize very well to the regression run of the application, and are therefore quite important to detect.

In order to address the preceding two concerns in a unified framework, we first use Algorithm 1 to generate short patterns using only the "broad-brush" notions of high-variance, without distinguishing between tagged nodes using their statistics (mean, CoV). Then we group patterns with identical call-contexts (arising from different tagged nodes) and use *pattern-similarity-trees* (PST) to start differentiating between them based on their statistics. The initial group forms the root of a PST. We apply a *Similarity-Measure* (SM) function on the group to see if it requires further differentiation. If the patterns in the group have widely different means or CoVs, and the programmer wants this to be a differentiating factor, then the similarity check with the appropriate SM will fail. In our experimental evaluation, we use an SM that checks if the corresponding means and CoVs of the two patterns being compared are within 10% of each other; the programmer can choose to plug in a different SM, say one that checks only on means.

Once the SM test fails on a group, all the patterns in the group are extended by one more parent function from their corresponding call-chains (tagged CCT nodes are kept associated with patterns they generate). This will cause the resulting longer patterns to start to differ from each other. Again identical longer patterns are grouped together as multiple children groups under the original group. This process of tree-subdivision is continued separately for each generated group until the SM function succeeds in all current leaf nodes. At this point, each of the leaf groups in the PST contain one or more identical patterns. The patterns across different leaf groups are however guaranteed to be different in some part of their prefixes. And patterns in different leaf groups may be of different lengths, even though the corresponding starting patterns in the root PST node were of the same length. All the identical patterns in the same leaf-node are collapsed into a single detection-pattern. For example, an SM function that differentiates on σ but not on means (or only weakly on means), will produce leaf nodes that contain patterns with a single σ but a collection of widely varying means.

3.4 Ranking Impact of Patterns

The previous steps produce numerous patterns (11 to 46 patterns for our benchmarks) characterizing the variability in the application at multiple levels. It is highly desirable to rank the patterns in order to focus the programmer's attention on the ones that are most likely to contribute variability to the program. For this purpose we introduce a metric that we call the *Variability Impact Metric* or

`VIM`. The Chebyshev inequality introduced earlier points us towards a suitable definition for `VIM`. While the CoV $= \frac{\sigma}{\mu}$ indicates whether the variations are large with respect to the mean, the $k\sigma$ term in the Chebyshev inequality indicates the absolute amount of variability. The variability per invocation multiplied by the total invocation count of that pattern gives the total amount of variability contributed by the innermost function in the pattern to its immediate parent. Therefore, we define `VIM` as follows, with N being the invocation count of the innermost function in the pattern:

$$\texttt{VIM} = k\sigma N \tag{2}$$

While this metric indicates how much variance is contributed by the innermost function F to its immediate parent C (referring to the pattern in Figure 2), it is not necessarily implied that the pattern's variance contribution propagates up the call-chain to A or B. For example, if B invokes C from inside a loop, then the `VIM` for C will measure the variance impact to iterations of the loop, not to B directly. In fact, it is possible that B is not variant at all if each iteration of C consumes correspondingly lower time if the loop-count is high, and vice-versa when the loop-count is low, leading to a constant execution-time for the loop across all invocations of B. A similar situation can occur without loops if B invokes C inside a very infrequently executed branch.

Despite the limitation described above, the profile analysis technique is excellent for *eliminating* unlikely contributors of variance. Therefore, the correct way to interpret the produced patterns is to think of them as *highly likely contributors* of variance. This immediately allows the programmer to narrowly focus on very limited parts of the application in order to identify the causes of violations to the soft real-time requirements. The programmer would of course have to examine relative invocation counts along a given pattern's call-chain to infer how far up the call-chain an innermost function is likely to be contributing variance.

4 Experimental Evaluation

Table 1. Patterns Found in Benchmarks

Benchmark	Profiling on `D1`: Pat. Generation				Regression on `D2`			
	# of steps	*Pass time (seconds)*	*# of patterns*	*Pattern Set size*	*# of steps*	*Pass time (seconds)*	*Pattern Set size*	*Pat Set overlap*
H.263enc	30000000	397	9	7	60000000	1245	7	100%
H.263dec	25000000	341	30	5	60000000	2194	6	100%
findTux	30000000	402	60	3	40000000	2833	3	100%
mpeg2enc	30000000	387	44	5	60000000	2943	5	100%
mpeg2dec	30000000	402	20	5	60000000	1657	5	80%

The Statistical Analyzer tool has been written in Python. We did not use any high-performance numerical or scientific libraries (such as NumPy, SciPy) in the Python implementation. We profile instrumented a number of applications in the MediaBench II Video suite and a real-time object-recognition benchmark (mimas-findTux) from the Mimas Computer Vision application-suite [18]. We generate profile data (sequence of profile events) for each benchmark using the input data sets provided with the benchmark suites, or some larger external data sets if the profiles are too short. Specifically, we use two different input data-sets for each benchmark, referred to as `D1` and `D2`.

We run profile analysis on `D1` to create patterns and then use `D2` for the regression run that we use to validate the statistics of the patterns found previously. The regression run simulates the application call-stack using the profile events. No `CCT` is constructed and no analysis is performed in the regression phase. We use a generic finite-state-machine sequence detector to detect the occurence of the patterns at the top-of-the-stack. Such a sequence detector needs to check the call-stack for the possible occurence of every patterns on every profile event. This is the cause of the significant slow-down seen in Table 1 in the pass times for the regression runs compared to the profile runs. *We would like to emphasize that the profile analysis time consists entirely of the time to read and parse the profile file from disk. The actual time for all of the analysis combined (variance tagging, minimum call-context detection, etc) consumes a fraction of a second.*

Table 1 shows the length of the `D1` profile (in terms of number of `entry` / `exit` events) used to generate patterns, the number of high-variance patterns found, and the length of the `D2` profile used during regression to simulate the real-world execution of an application. The *Pass Time* refers to the duration of time needed to complete profile analysis or regression.

Clearly during regression the input data set is different, which will lead to corresponding changes in the call-chains invoked, their frequencies and their variant behaviors. However, in our validation we strive to demonstrate that the patterns capture the statistical behavior of the application at a more fundamental level, which tends to remain relatively constant across different data-sets. In order to demonstrate this, we introduce the notion of a *Pattern Set* both for Profiling and Regression. We define the Pattern Set to consist of a subset of patterns that are found to be most impactful as measured by their Variability-Impact-Metric (`VIM`). Specifically, we limit the Pattern Set to only those patterns whose `VIM` is atleast 10% of the `VIM` of the pattern with the highest `VIM`. This is done separately for Profiling and Regression, leading to the construction of two potentially disjoint sets. Table 1 shows that in fact the Regression Pattern Set very closely mirrors the Profiling Pattern Set (*Pattern Set Overlap* column). This implies that the same set of patterns that were found to be most impactful during Profiling tend to remain most impactful during Regression. The Pattern Set spans an order-of-magnitude of the largest `VIM` values (i.e., $10\times$). We chose to define the Pattern Set as such because we expect the data-set induced variations to cause relative fluctuations due to changes in length of data (number of events) and type of data (for example, encoding video with constant background versus moving background, different frame-dimensions, etc). Despite these variations in characteristics of input data, the most impactful patterns found on `D1` tend to remain most impactful on `D2` as well, validating our intuition that our patterns capture variant behavior in a statistically sound manner. In mpeg2dec, the `VIM` of one pattern was just slightly smaller during regression causing it to be dropped from the Regression pattern set. Similarly, a pattern that had barely missed inclusion in the Profiling pattern set got included in the Regression pattern set. However, both these patterns have similar `VIM`s (in the order-of-magnitude of sense). Therefore, despite a Pattern Set Overlap of only 80%, this result also shows that Profiling and Regression Pattern Sets match closely for mpeg2dec. In H.263dec, there was a pattern that barely missed inclusion in the Profiling Pattern Set, but got included in the Regression Pattern Set.

Figure 3 shows the distribution of the mean and CoV values for all the patterns discovered, on a per-benchmark basis. For each benchmark, the left-segment in the scatter-plot shows the distribution found during Profiling (on `D1`), and the right-segment shows during Regression (on `D2`). No `VIM` based distinction is made between patterns; the least varying pattern with low invocation-count is given a point just like the most impactful pattern. For all benchmarks the distributions between Profiling and Regression are very similar, except for a uniform linear shift and uniform scaling of one distribution with respect to the other. When we look at Figure 4 plotted using only patterns in the Profiling and Regression *Pattern Sets*, we again see a close similarity between Profiling and Regression distributions, indicating that the dominant patterns are fundamentally associated with the application behavior, regardless of data-sets. For example, encoding raw video with a larger image frame-size quadratically increases the mean and possibly the CoV of a motion-estimation pattern, but motion-estimation remains dominant independent of the image frame-size.

The following is representative across the benchmarks of the *compaction of information* achieved in going from raw profile data to the final profile results: 800MB to 1.3 GB of raw profile event data reduced to a `CCT` with 600 to 800 nodes, out of which 200 to 350 nodes were found significant, out of which 16 to 116 nodes were tagged high-variant, which were grouped down to 9 to 60 patterns with identical contexts and similar means and CoVs (using pattern similarity tree), finally out which 3 to 7 were dominant patterns (pattern set).

4.1 Case Study: H.263enc

Figure 5 shows the Profiling Pattern Set for the H.263enc benchmark, sorted from the most impactful to the least. The `VIM` found for each pattern is shown for the Profiling and Regression phases. Function-names are shown in boxes and the edge-annotations give the `lexical-id` (lexical position) of the call-site of the callee (sink of arrow) within the body of the caller (source of arrow). The italicized number on top of each box gives the number of times the corresponding function was invoked as part of the pattern. A pattern's invocation-count corresponds to the invocation-count of the function in the left-most box. This is the innermost function of the pattern, and the entire pattern occurs only when the entire call-chain segment occurs on the stack. Therefore, the invocation-count of the innermost function is the invocation count of the pattern.

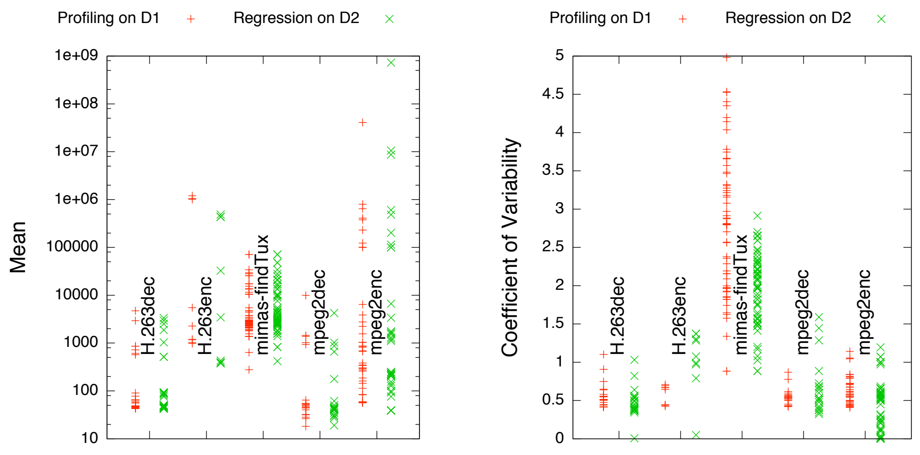


Fig. 3. Comparison of *mean* and *CoV* scatter-plots between Profiling `D1` and Regression `D2` using **all patterns**

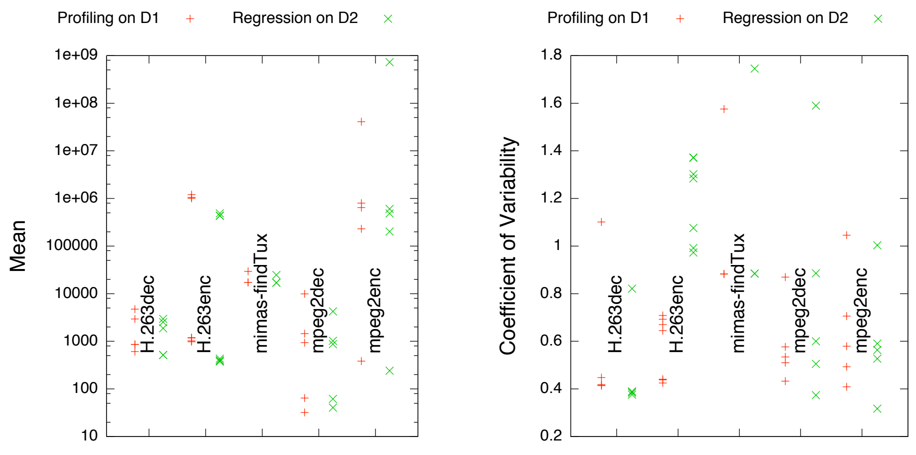


Fig. 4. Comparison of *mean* and *CoV* scatter-plots between Profiling `D1` and Regression `D2` using **Pattern Set**

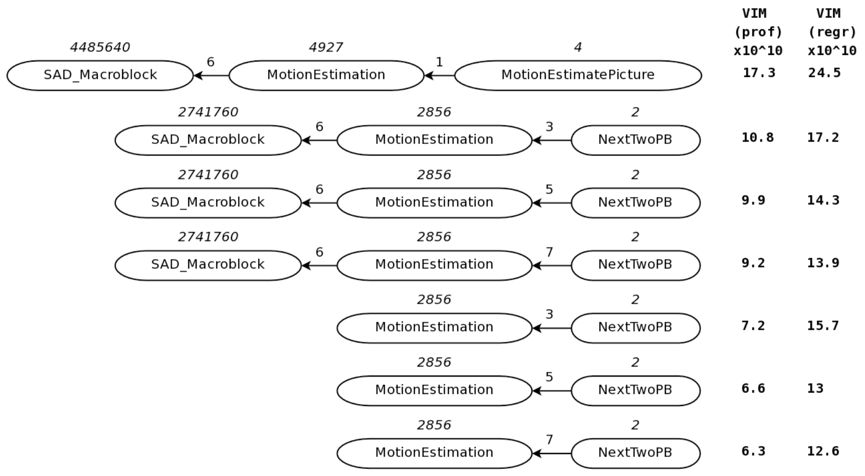


Fig. 5. Pattern Set for H.263enc

The patterns in Figure 5 were automatically discovered by the profile analysis framework with no guidance from the user, and no application or domain knowledge. Yet, these patterns closely mirror conventional wisdom about the parts of video-encoding applications that are the most important with regards to meeting or violating soft real-time requirements. Motion-estimation related macroblock search-spaces are known to be the most variant parts of video encoding [16], since the search space can be quite large and it is hard to know up front how quickly the search will terminate.

Note that the middle three patterns and the bottom three patterns are identical except for a difference in `lexical-ids`. In both cases, the multiple identical patterns have very similar statistical characteristics (`VIM`s and also from their positions in the scatter plots). These could have been combined into a single pattern in both cases, but our analysis framework distinguishes based on `lexical-ids` within patterns. The downside here is having three patterns where one would suffice, but in general this produces greater resolving power between identical call-chains whose behavior varies between call-sites, such as with mpeg2enc.

5 Related Work

Existing application profiling techniques look for program hot-spots and hot-paths [4, 5, 9]. These techniques attempt to find performance bottlenecks in an application, and do not attempt to identify variant behavior impacting an application's soft real-time characteristics.

Calder et al. have used statistical techniques to characterize large scale program behavior using few recurrent intervals of code [7] and to find phase change points in the dynamic execution of a program [8]. However, their work was not intended for mining soft real-time characteristics of an application, and cannot be adapted for such. In particular, they seek out intervals in [7] with closely matching set of dynamic basic-blocks, whereas we seek out call-contexts where the same function exhibits highly variant execution time.

Variability Characterization Curves (VCCs) and Approximate VCCs [10] have been used to characterize the variability in the workloads of multimedia applications. Such analysis techniques require domain-specific knowledge of the application before they can be applied. Similarly, there are custom techniques for improving the QoS of each type of application, such as by Roitzsch et al [15] that develop a higher-level representation model of a generic MPEG decoder, and based on this predict video decoding times with high accuracy. In contrast, our framework characterizes the variant behavior in the application in a completely domain-independent manner, with no assistance from the user.

For applications written using real-time constructs/formalisms such as tasks and deadlines, there is an established body of formal techniques [11, 12] that analyze or ensure the real-time characteristics of the application. For monolithic applications written without the use of these abstractions, our framework is unique in its ability to characterize their soft real-time behavior.

Worst-Case-Execution-Time (WCET) [13] is an analysis methodology applicable to monolithic applications, and has been incorporated into commercial products such as from AbsInt [17]. However, for non-safety-critical, compute-intensive applications like gaming and video, knowledge of the *likely range* of real-time behavior is more important for driving design optimization than knowledge of worst case behavior. The likely range (detected by our technique) can be substantially removed from the worst case, thereby diminishing the value of characterizing the worst case behavior for such applications.

In contrast with prior work [14] on identifying variant behavior in monolithic applications, the techniques in this paper establish statistically robust probability bounds on variant behavior and produce concise results prioritized by their impact on soft real-time behavior.

6 Conclusion

In this paper we demonstrated that analyzing a profile sequence of time-stamped function entry and exit events can be used to *i)* identify the dominant soft real-time components of functionality in an application, *ii)* determine the context-sensitivity of the behavior of the identified components, and *iii)* concisely convey the components and their context sensitivity to the programmer using patterns consisting of minimal-length call-chain segments. Further, we established that the dominant patterns detected during profile analysis continue to remain dominant in regression runs of the application on input data sets that have different characteristics (differences in frame-dimensions, encoding format, degree of motion in input videos). This experimental validation coupled with a sound foundation of our algorithms in statistical theory suggests that our analysis detects fundamental aspects of an application. Lastly, we find that patterns identified by our profile analysis in well-known multimedia applications correspond closely with extensive prior research studying the causes of soft real-time variance in these applications. In conclusion, our technique concisely captures the true soft real-time characteristics of a monolithic application, for which existing real-time analysis techniques are inapplicable.

Future Work Call-chain segments were found useful in defining contexts for components, but call-chains do not sufficiently capture which components contain other components, and more importantly, whether contained components contributed significant variance to any parent component. We are currently developing a concise hierarchical representation for capturing the cause-effect and containment structure between components.

References

1. Isovic, D., Fohler, G., Steffens, L.: Timing constraints of MPEG-2 decoding for high quality video: misconceptions and realistic assumptions. ECRTS '03
2. Ammons, G., Ball, T., Larus, J. R.: Exploiting hardware performance counters with flow and context sensitive profiling. PLDI '97
3. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. CGO'04
4. Duesterwald, E., Bala, V.: Software profiling for hot path prediction: less is more. SIGPLAN Not. 35, 11, 202–211 (2000)
5. Hall, R. J.: Call path profiling. ICSE '92
6. Freund, J. E., Walpole, R. E.: Mathematical statistics (4th ed.)
7. Sherwood, T., Perelman, E., Hamerly, G., Calder, B.: Automatically characterizing large scale program behavior. SIGOPS Oper. Syst. Rev. 36, 5, 45–57 (2002)
8. Lau, J., Perelman, E., Calder, B.: Selecting Software Phase Markers with Code Structure Analysis. CGO '06
9. Arnold, M., Hind, M., Ryder, B. G.: Online feedback-directed optimization of Java. OOPSLA '02
10. Liu, Y., Chakraborty, S., Ooi, W.T.: Approximate VCCs: a new characterization of multimedia workloads for system-level MpSoC design. DAC '05
11. Lin, C., Brandt, S. A.: Improving Soft Real-Time Performance through Better Slack Reclaiming. RTSS '05
12. Wandele, E., Thiele, L.: Abstracting functionality for modular performance analysis of hard real-time systems. ASP-DAC '05
13. Gustafsson, J., Ermedahl, A., Sandberg, C., Lisper, B.: Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution. RTSS '06
14. Kumar, T., Sreeram, J., Cledat, R., Pande, S.: A profile-driven statistical analysis framework for the design optimization of soft real-time applications. ESEC-FSE '07
15. Roitzsch, M., Pohlack, M.: Principles for the Prediction of Video Decoding Times Applied to MPEG-1/2 and MPEG-4 Part 2 Video. RTSS '06
16. Girod, B., Steinbach, E., Färber, F.: Performance of the H.263 Video Compression Standard. J. VLSI Signal Process. Syst. 17, 2–3 (1997)
17. http://www.absint.com
18. http://www.shu.ac.uk/research/meri/mmvl/research/mimas/