A Profile-Driven Statistical Analysis Framework for the Design Optimization of Soft Real-Time Applications

Tushar Kumar, Jaswanth Sreeram, Romain Cledat, Santosh Pande
College of Computing
Georgia Institite of Technology
tushark@ece.gatech.edu, {jaswanth, romain, santosh}@cc.gatech.edu

ABSTRACT

Soft real-time applications lack a formal methodology for their design optimization. Well-established techniques from hard real-time systems cannot be directly applied to soft real-time applications, without losing key benefits of the soft real-time paradigm.

We introduce a statistical analysis framework that is well-suited for discovering opportunities for optimization of soft real-time applications. We demonstrate how programmers can use the analysis provided by our framework to perform aggressive soft real-time design optimizations on their applications.

The paper introduces the Context Execution Tree (CET) representation for capturing the statistical properties of function calls in the context of their execution in the program. The CET is constructed from an offline-profile of the application. Statistical measures are coupled with techniques that extract runtime distinguishable call-chains. This combination of techniques is applied to the CET to find statistically significant patterns of activity that i) expose slack in the execution of the application with respect to its soft real-time requirements, and ii) can be predicted with low overhead and high reliability during the normal execution of the application.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques

General Terms

Design, Performance, Measurement

Keywords

Statistical Analysis, Profiling, Pattern Detection, Signatures, Behavior Prediction, Soft Real-time

ESEC/FSE'07, September 3–7, 2007, Cavtat near Dubrovnik, Croatia.
ACM 978-1-59593-812-1/07/0009.

1. INTRODUCTION

Soft real-time applications are currently designed and optimized using rather ad-hoc means. A very large class of emerging applications fall under the category of soft real-time, including end-user applications like gaming and streaming multimedia (video encoders and decoders, for example). Such applications tend not to be mission-critical like hard real-time applications that require absolute guarantees that their execution deadlines be met. With hard real-time applications, guarantees on meeting deadlines can be made by following very conservative design principles with provable properties [3], or by having a runtime system that conservatively schedules the component tasks of the application to ensure that certain real-time guarantees are met [7]. In contrast, soft real-time applications do not require absolute guarantees that their real-time constraints will always be satisfied. Most soft real-time applications only require that their deadlines are met most of the time, within certain statistical bounds. For example, a game or video decoder might require an *average* frame-rate of 30 frames-per-second, with 95% probability that the instantaneous frame-rate does not deviate from this average by more than 5 frames-per-second. Such relaxation of guarantees allows a soft real-time application to aggressively perform more sophisticated computation and maximally utilize the available compute resources.

In contrast with hard-real-time systems where a large body of modeling and scheduling work exists [3], soft real-time applications can use such a very wide variety of *relaxed* guarantees that so far no sufficiently broad formal framework exists for the analysis and design of these applications. The main problem is the lack of tools that could assist a soft real-time system designer in gaining the necessary insights to tighten the application design.

This paper introduces a framework for the analysis of soft real-time applications. The framework finds dominant or recurring patterns of behavior from the profile of an application, and produces signatures (detection patterns) that can be used to *i) identify* the occurence of these behaviors in future runs of the application, *ii) predict* the occurence of these behaviors sufficiently in advance, and *iii)* make *statistical assertions* about the likelihood of the occurence of these behaviors during the application's execution. The programmer can use these patterns to assess how well the application will satisfy various relaxed real-time deadlines.

1.1 Related Work

Existing application profiling techniques look for program hot-spots and hot-paths [6, 1] such as basic-blocks, loops or

functions exhibiting high execution-time. Certain dynamic optimization techniques benefit from detecting hot paths at runtime and dynamically optimizing code [4, 2]. None of these techniques however *apriori* characterize the soft real-time characteristics of an application. Work has been done by Calder, et al [9] that captures statistical information (like standard deviations) along function call-graphs and control-flow-graphs. However, the call-graph and control-flow-graph representations do not adequately capture the multiple contexts in which functions can be invoked, and force the contexts to be the edges of the call-graph and control-flow-graphs. In contrast, our framework explicitly distinguishes function-call invocations based on their dynamic context of invocation (call-stacks), which can be much more varied. Subsequent analysis selectively merges information across multiple contexts if they exhibit the same behavior. Furthermore, our framework is unique in allowing programmers to incorporate in their applications a low-overhead mechanism for the detection and prediction of patterns generated during offline profile analysis of the applications.

Our analysis techniques are statistical in nature and are robust against "outlier" behaviors that disrupt otherwise well-formed patterns of activity. Other techniques, like Whole Program Path analysis [8], are susceptible to disruption in the presence of extraneous intervening program activity during the detection of a broader pattern of activity.

1.2 Overview of Analysis Framework

Our framework first profile-instruments a C application. It then runs our Statistical Analyzer tool that detects patterns of behavior and generates prediction patterns and statistical guarantees for those. The patterns of behavior consist of segments of function call-chains, annotated with the statistics predicted for them. The call-chains are further refined into *minimal distinguishing call-chain sequences* that unambiguosly detect the corresponding pattern of behavior when it starts to occur at runtime, and make statistical predictions about the nature of the behavior.

We introduce the Context Execution Tree `CET` representation of the profile information, and various analysis techniques that can identify behavior patterns along with associated statistical guarantees based on the `CET`.

2. CONTEXT EXECUTION TREE

We introduce the Context Execution Tree (`CET`) representation for capturing the dynamic context of execution of function-calls in a program, observed during a given execution of the program. Nodes in the `CET` represent function invocations (calls) during the execution of the program. The root node represents the invocation of the `main` function of a C program. For a given node, the path to it from the root node captures the sequence of parent function calls present on the program call-stack when the function corresponding to the node was called. Multiple invocations of a function with the same call stack will all be represented by a single node. However, multiple invocations of the same function with different call-stacks will result in multiple nodes for the same function, with the path from the root to each node capturing the corresponding call stacks.

The `CET` can be constructed from a profile of program execution. The profile consists of a sequence of function-entry and function-exit events in the order of their occurence during the execution of the program.

Figure 1 illustrates the structural properties of an example `CET` constructed from the adjacent program. Function `A` was invoked from two call-sites within the parent function `P`. This leads to two children nodes for function `A`. Since function `B` was never invoked from the left `A` node, it only gets a *NULL* edge under the `A` node at the lexical position of its call-site in the body of function `A`.

2.1 Node Annotations

Each `CET` node is annotated with the following pieces of information 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.

In Figure 1, X_i represents the sequence of execution-times observed at each node. The *std* and *cov* annotations show the standard-deviation and coefficient-of-variability metrics calculated over the entire run of the application.

2.2 CET Construction Procedure

The Statistical Analyzer constructs the CET (the tree structure) in a single pass over the profile sequence. It makes a second pass to calculate the variance node annotations. The profile sequence consists of the sequence of function-call entry and function-call exit events along with the corresponding time-stamps. We use the dynamic instruction count since the start of the program as the time-stamp for each event.

3. DETECTING PATTERNS OF BEHAVIOR

Once the `CET` has been constructed and its node annotations calculated, the `CET` is traversed in pre-order to determine nodes which exhibit interesting behavior as evidenced by their node annotations. Nodes whose total execution time constitutes a miniscule fraction (say, $< 0.02\%$) of the total execution time of the program, are deemed as *insignificant* and excluded from all further analysis. All other nodes are deemed *significant*. Since `CET` nodes subsume the execution time of their children nodes, once a node is found to be insignificant, the nodes in its children subtrees are guaranteed to be insignificant as well.

3.1 Tagging Nodes

We examine the annotations of nodes to determine if the corresponding nodes exhibit one or more of the following types of interesting behavior:

1. **low-variance**: The variance σ^2 is *low*.
2. **high-variance**: The variance σ^2 is *high*.

In order to provide an unambiguous, program-independent basis for comparison of variant behavior, we use the *Coefficient-of-Variability* (CoV) metric [5] on each node. This is defined as follows:

$$CoV = \frac{\sigma}{\bar{X}}$$

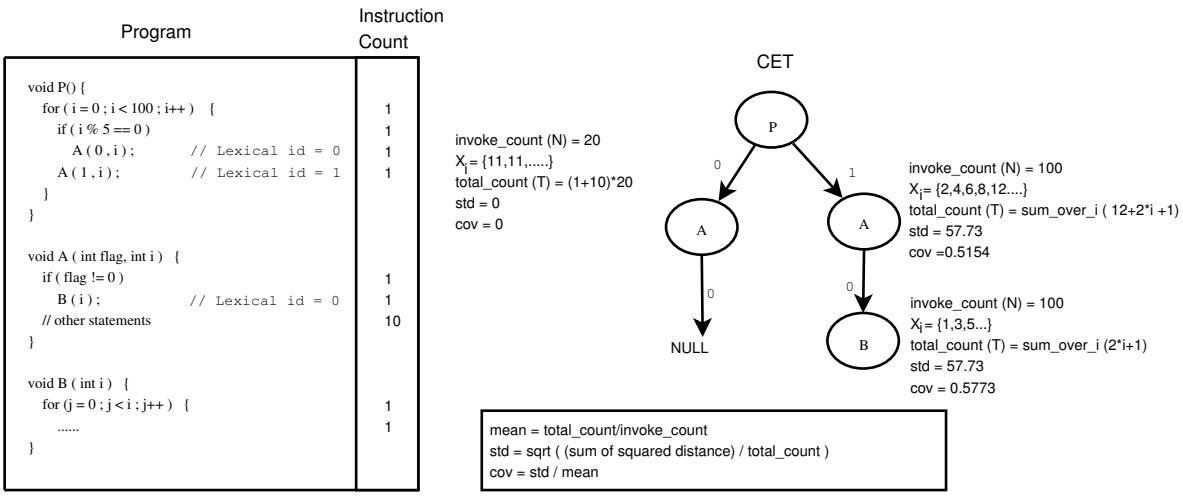

Figure 1: Sample Program, CET and Annotated Node Statistics

Once the CET is constructed from the profile data, it is traversed in pre-order and using CoV values individual nodes may be *tagged* as being *low-variant* or *high-variant*. As mentioned earlier, the traversal is restricted to significant nodes.

3.2 Signature Generation for Patterns

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 interesting 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 between `main` (the CET root node) and the tagged node.

The sequence of names of function-calls in the call-chain segment becomes the detection pattern arising from the tagged node. This particular detection pattern might occur at other places in the significant part of the CET. Quite possibly, the occurence of this detection pattern elsewhere in the CET does not lead to the same interesting statistical behavior that was observed at the tagged node. Therefore, our key criteria in generating the detection pattern is the following:

> All occurences in the significant CET of a detection pattern arising from a tagged node must exhibit the same statistical behavior as 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 now cannot occur anywhere else in the CET. In many applications, patterns that extend 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 CET. In most applications we expect that the statistical behavior of the function call at the top of the stack can be predicted using a short sequence of function-calls occuring just below it in the call-stack, without going all the way to `main`. In this paper we direct our attention towards detecting just such call-sequences. We call these *Minimal Distinguishing Call Sequences* (MDC) corresponding to any given 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 significant portions of the CET.

Given a tagged node P that is a call-instance for function F, the MDC is constructed by extending the parent call-chain starting at P until the call-chain is just long enough to be different from a call-chain of the same length originating at any other significant instance of F in the CET, that does not match the statistical behavior of P. Significant instances of F that match the behavior at P and have identical distinguishing call-contexts are detected by the same MDC, effectively merging them with P.

3.3 Scheme for Detecting Patterns at Runtime

The application code can be easily modified by the programmer to incorporate the detection of specific MDC sequences that the programmer determines as being most useful to detect. Given a MDC sequence the programmer has to instrument the function-calls that occur in it. If the MDC sequence is a call-chain of length k, then let $MDC[0]$ denote the uppermost parent function-call, and $MDC[k-1]$ denote the function-name of the tagged node that generated this MDC sequence. Therefore, the pattern will be detected to have occured if the $MDC[k-1]$ function is pushed at the top of the call-stack that already contains $MDC[k-2] \cdots MDC[0]$ function-calls just below in the stack. And over multiple occurences of this same pattern at runtime, the observed statistics are expected to match the behavioral statistics of the tagged node in the CET that generated this MDC sequence.

3.4 Early Prediction of Call-Chain Patterns

In the previous discussion, a pattern could only be detected at runtime whenever its corresponding call-chain segment occured *in full*. Only when the entire call-chain pattern occured on the call-stack, could a prediction about the execution time of the $MDC[k-1]$ function be made. However, with additional analysis, it is possible to observe the occurence of only a *prefix* of the pattern and predict with

Benchmark	Pass Time (seconds)						**Pattern Results**	
	Profile		Regression		Profile	Regression		
	Pass 1	Pass 2	Pass 1	Pass 2	Events	Events	Low-Var	High-Var
mpeg2enc	91	263	1877	1881	10000000	30000000	33 (33, 31)	24 (12, 21)
mpeg2dec	92	490	1584	1618	10000000	30000000	17 (13, 14)	31 (30, 31)
h263dec	44	254	1783	1809	5000000	25000000	32 (26, 29)	28 (26, 26)

Key to **Pattern Results**		
Pattern type	During Profiling	During Regression: observed characteristics of patterns
Low-Var	Number of such patterns detected	(patterns that remained low-variance, patterns with unaffected means)
High-Var	Number of such patterns detected	(patterns with unaffected σ, patterns with unaffected means)

Table 1: Patterns Found in Benchmarks

high probability that the remaining *suffix* of the call-chain pattern will occur (with the behavior statistics associated with the full pattern). This *prefix-suffix analysis* is done by examining each possible prefix of a pattern at a time. For a given prefix, the ratio of the occurence count of the full pattern in the `CET` against the occurence count of just the prefix serves as the *prediction-probability* that the corresponding suffix will occur in the future given that the prefix has been observed on the call-stack.

4. EXPERIMENTAL EVALUATION

The Statistical Analyzer tool is written entirely in python and did not use any high-performance numerical or scientific libraries (such as NumPy or SciPy). We used the LLVM compiler infrastructure to automatically profile instrument applications in the MediaBench II suite. Table 1 shows the number of profile events that were used to generate patterns, and the number of regression events that were used to simulate detection of the generated patterns in the real-world execution of the corresponding application. We produced a profile sequence for each benchmark using the input data sets provided with the benchmark suite, or some larger external data sets if such profiles were too short.

The Statistical Analyzer first uses a prefix of the profile (approx. 5 to 10 million events) to identify interesting statistical behaviors and generate detection patterns for an application benchmark. Then, the Statistical Analyzer reads the entire profile (approx. 25 to 30 million events) in the regression run of the application. The regression run simulates the application call-stack using the profile events. No `CET` is constructed and no analysis is performed during regression. 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 each pattern at every profile event, thereby causing the regression passes to be significantly slower than the corresponding profile passes (Table 1). However, the techniques we proposed for the actual runtime detection of patterns in applications (see Subsection 3.3) are quite lightweight as they require the instrumentation of only the functions and call-sites that occur in a pattern. Mean and variance statistics are calculated based on data collected for each pattern during regression. Table 1 shows the number of patterns of each type generated during profiling, and for how many patterns the regression statistics closely matched the statistics predicted by the profiling stage. The subtable titled "Key to Pattern Results" explains how to interpret the results under the "Pattern Results" columns. The results show that, for almost all patterns, the predicted statistics closely match those observed when the patterns are detected during the regression run.

5. CONCLUSION

We have demonstrated that the combination of statistical pattern classification techniques and distinguishing call-chain sequence extraction techniques provides a powerful framework for the analysis and design optimization of soft real-time applications. Our experiments show that the generated patterns and their associated statistics are generalizable to the real-world execution of the application.

6. REFERENCES

[1] Arnold, M., Hind, M., and Ryder, B. G. Online feedback-directed optimization of java. In *OOPSLA '02: Object-oriented programming, systems, languages, and applications* (2002), pp. 111–129.

[2] Bala, V., Duesterwald, E., and Banerjia, S. Dynamo: a transparent dynamic optimization system. In *Programming Language Design and Implementation 2000* (2000), pp. 1–12.

[3] Bernat, G., Colin, A., and Petters, S. Wcet analysis of probabilistic hard real-time systems, 2002.

[4] Burke, M. G., Choi, J.-D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M. J., Sreedhar, V. C., Srinivasan, H., and Whaley, J. The jalapeno dynamic optimizing compiler for java. In *ACM Conference on Java Grande 1999* (1999), pp. 129–141.

[5] Freund, J. E., and Walpole, R. E. *Mathematical statistics (4th ed.)*. 1986.

[6] Hall, R. J. Call path profiling. In *ICSE '92: Proceedings of the 14th international conference on Software engineering* (1992), pp. 296–306.

[7] Kavi, K. M., Youn, H. Y., Shirazi, B., and Hurson, A. R. A performability model for soft real-time systems. In *27th Hawaii International Conference on System Sciences* (1994), pp. 571–580.

[8] Larus, J. R. Whole program paths. In *Programming Language Design and Implementation 1999* (1999), pp. 259–269.

[9] Sherwood, T., Perelman, E., Hamerly, G., and Calder, B. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev. 36*, 5 (2002), 45–57.