Dominant Variance Characterization

Tushar Kumar
School of Electrical & Computer Engineering
Georgia Institute of Technology
tusharkumar@gatech.edu

Santosh Pande
College of Computing
Georgia Institute of Technology
santosh@cc.gatech.edu

ABSTRACT

There are a whole range of program analysis techniques that characterize different aspects of an application's performance: hot-spots, distinct phases of behavior, code segments that could potentially run in parallel, etc. For a growing class of applications, there is a need to add another analysis technique to the repertoire that can characterize the locations and underlying causes of execution time variance in repetitive parts of the application.

In this paper we introduce the notion of dominant variance analysis of an application. We illustrate the unique performance optimization benefits of performing such an analysis. We motivate that traditional program analysis and profiling techniques are not sufficient to analyze the variant execution time behavior of the application. We introduce a new program representation called Variance Characterization Graph that is used both as the intermediate representation to enable the dominant variance analysis and as the final representation that provides concise and actionable information to programmers. We identify the unique challenges associated with characterizing the dominant behavior of an application and develop a methodology based on statistical pattern matching to efficiently recognize dominant patterns of behavior.

1. INTRODUCTION

Programmers need to understand program behavior in order to tune the performance of their applications. This understanding is typically gained by profiling the application on representative data sets. The type of behavior examined during profile analysis dictates how the profiling results can be used by the programmer for performance tuning.

Most traditional profiling techniques are oriented towards minimizing overall program execution time, such as profiling for hot-spots or hot-paths. Other techniques detect phases in an application's execution where each phase has distinct characteristics with regards to stall cycles, cache miss rates and instructions per cycle executed. Another set of techniques called Worst Case Execution Time analysis, attempt to place bounds on the execution times of components in safety-critical applications. All these techniques aid the programmer to debug different aspects of an application's performance. One aspect of performance that is not covered by existing techniques is the characterization of the *variation* in execution time exhibited by components in the application. The benefits of such characterization include determining whether an interactive application can be expected to be responsive to the user, determining how much speedup is likely in parallelizing sections of a sequential application, or detecting if a security application is vulnerable to attacks that guess the underlying dynamic control-flow based on observed variations in its execution time.

1.1 Analyzing for Variance

We seek to understand the behavior of an application in terms of the variations exhibited in the execution time of functions. While traditional profiling techniques often suffice for understanding the average behavior of every function, the patterns of variation in the execution time of functions are much more revealing about the execution characteristics and functional design of the application. Analyzing on variance allows long range relations to be revealed between groups of functions whose behavior varies in synchrony. Further, analyzing on variance rather than hot-spots reveals where the program performs "interesting" processing of data, since clearly at these functions the nature of the data significantly affects the type and quantity of computation performed. This has the potential to reveal the implicit functional/high-level design of the application even when analyzing in the absence of application or domain specific knowledge. Ultimately we would like to identify groups of functions whose variant behaviors are related, and identify the dominant modes of behavior exhibited collectively by a group. In particular, we would like to quantify relationships between the functions where high variance is observed and other functions that are the principal underlying causes of the observed high variance, exposing the *variance contribution structure* of the application. The overarching intent here is to provide the programmer with a succint picture of the application-wide variant behavior, including cause-effect relationships. At the same time, the analysis results need to provide actionable information that allow the programmer to readily identify where in the application code fixes need to be made to bring about a desired change in application wide behavior.

While analyzing variant behavior has great potential for a deeper understanding of program behavior as outlined above, there are significant challenges to be overcome in order to realize this potential. Consider the space of possible ways to group functions, ways to summarize a continuum of a function's behavior into a few discrete modes of behavior, ways to filter which functions are useful for understanding application behavior and which can be omitted. This space of possibilities grows combinatorially. Further, choices that appear suitable over one part of the profile data may be poorly suited over other parts of the profile. This is further complicated by the fact that the raw profile data over which the analysis is to be performed can easily exceed billions of profile events, making it untenable to retain the raw profile data in memory or even to perform random seeks on it from disk. This precludes any naive approach that exhaustively explores the space of possibilities looking for a good solution. Instead, we rely on recognizing specific properties about the *dynamic call-structure* of the application, the *mathematical nature of variance*, and the power of *statistical clustering techniques* in order to make the analysis tractable yet effective.

1.2 Contributions

We make the following contributions to the state of the art:

- We motivate the optimization potential offered by the characterization of variant behavior in an application's functions. Further, we motivate that variance can be much more powerfully characterized when the context sensitivity of behavior is taken into account, producing correspondingly greater opportunities for program understanding and optimization.
- We propose a program representation (Variance Characterization Graph) that succintly captures the dominant variant behavior exhibited by the application. The properties of the representation allow it to both highly summarize similar behavior observed across the application's call structure, as well as precisely contrast the locations and differing nature of variant behavior.
- The VCG representation allows the programmer to easily and unambiguously map observed behavior both to the lexical code locations in the program, as well as map the behavior to the dynamic call-structure of the application.
- We show how the mathematical nature of variance, specific properties of the dynamic call structure of an application and statistical clustering technique can be used to derive appropriate metrics and algorithms that construct and operate on the VCG

representation.

2. VARIANCE IN A FUNCTION

Consider a function called F in an application. Let G_0, G_1, ...G_k be functions invoked directly within the body of F. The invoked functions may be conditionally invoked in an if-statement or case-statement, or they may be repeatedly invoked within a for-loop, as illustrated in the C code example in Listing 1.

Listing 1: Example function F

```
void F(int R, int T) {
  G1();
  ...
  if(...)
    G2();
  for(i=0; i<R; i++) {
    G3();
    S; //local code statements
    if(T >= 0)
      G4();
    else
      G5();
  }
}
```

For our labeling purposes, the functions are arranged in the lexical order G_1 to G_k within the body of F. Let X be a random variable representing the execution time of any *single* invocation of F. Let the random variable Y_i represent the *cumulative* execution time of G_i within a single invocation of F. Y_i could be zero if G_i was not invoked by F, or it could represent the total execution time over multiple invocations of G_i if G_i was invoked within a loop. Let Y_0 represent the *local* execution time of F, that is the part of the execution time of F that was not spent inside any children calls G_i. We choose a convention where Y_i includes the caller-side overheads of invoking G_i. Therefore, the following must hold between the observed values of random variables X and Y_i's for any given invocation of F.

$$X = \sum_{i=0}^{k} Y_i \tag{1}$$

Let $\bar{X} = E[X]$ represent the *mean* execution time of F over N invocations of F. Let $\sigma_X^2 = E[(X - \bar{X})^2]$ represent the *variance* in the execution time of F over these invocations. Let $\mathbf{C}$ represent the *co-variance matrix* between the random variables Y_0 to Y_k observed over these invocations.

$\mathbf{C}$ will be a $(k+1) \times (k+1)$ dimensional matrix, with $C_{i,j} = E[(Y_i - \bar{Y}_i)(Y_j - \bar{Y}_j)]$, again with the expectation computed over the N invocations of F. A covariance matrix is always *symmetric*, i.e., $C_{i,j} = C_{j,i}, \quad \forall i, j$.

Then, due to X being a sum over random variables Y_i, the following holds:

$$\begin{aligned} \sigma_X^2 &= \sum_{i=0}^{k}\sum_{j=0}^{k} C_{i,j} \qquad (2) \\ &= \underbrace{\sum_i C_{i,i}}_{\text{(self terms)}} + \underbrace{2 * \left(\sum_{i<j} C_{i,j}\right)}_{\text{(cross terms)}} \end{aligned}$$

Here we make no assumptions about the correlation between any r.v.'s Y_i and Y_j. Using Eq 2, the on-diagonal terms, $C_{i,i} = E[(Y_i - \bar{Y}_i)^2]$ may indicate whether invocations of the G_i function call within F contributes significantly to σ_X^2. For example, in Listing 1, if F is invoked with very different values for parameter R, the trip-count of the contained loop will vary accordingly. Correspondingly, Y_3 (the cumulative execution time of G_3 per invocation of F) will show significant variation over the N invocations of F, causing $C_{3,3}$ to have a large positive value. This would be true simply due to the variation in the trip-count of the loop, even when G_3 took constant time. Further, if the statements denoted by S dominate in Y_0 (the local execution time of F), then we can expect that Y_0 and Y_3 would show significant positive correlation, leading to a large positive value for $C_{0,3}$.

In contrast, if F is getting invoked with both positive and negative values for parameter T, then both Y_4 and Y_5 will show significant variation over the multiple invocations of F, even if the trip-count parameter R is kept fixed. This will cause $C_{4,4}$ and $C_{5,5}$ to have large positive values. However, the cross-terms $C_{4,5} = C_{5,4}$ will have large magnitude negative values indicating a strong negative correlation. Therefore, in order to determine which G_i's are the major causes of the variance in F, we need to not just determine if the corresponding $C_{i,i}$ terms significantly contribute to σ_X^2 via Eq 2. Additionally, we also need to examine if other G_j's are diminishing or enhancing the contribution through $C_{i,j}$ terms. Ultimately, we can expect to identify a *minimal* set of children functions $\{G_{i_1}, G_{i_2}, ..., G_{i_l}\}$ that together contribute significantly to σ_X^2 through all involved self-terms and cross-terms: $\sum_{i,j \in \{i_1,i_2,...,i_l\}} C_{i,j}$. This indicates that the $\{G_4, G_5\}$ children set in the current example would not contribute significantly due to large negative-valued cross-terms cancelling out the large positive self-terms. In particular, if G_4 and G_5 consume similar execution times, then the sum $Y_4 + Y_5$ will remain about constant for each invocation of F even though Y_4 and Y_5 may individually vary significantly over invocations of F. Instead, this example is likely to have $\{G_1\}$, $\{G_2\}$ and $\{G_3\}$ as separate sets of children that individually contribute significant variance to F. It is important to note that a G_i may not be a hot-spot within F, yet it may be a significant cause of the variance of F. In other words, $\bar{Y}_i$ (averaged over the N invocations of F) may be small, yet at the same time $C_{i,i}$ could be large. *This indicates that traditional hot-spot analysis usually does not help much with understanding variant behavior and its main causes.*

An important contrast to draw at this point would be to examine the variant behavior of G_3 as a stand-alone function versus its behavior observed under F. For simplicity of explanation, let's assume that F is always invoked with a fixed value for parameter R, and that G_3 is not invoked from any other call-site in the program. Then $N^{G_3} = R \times N^F$. N^{G_3} and N^F represent the invocation counts of G_3 and F respectively. Now, each Y_3^F will be the sum of the R observations of the X^{G_3} random variable corresponding to the R invocations of G_3 in each invocation of F. Since Y_3^F can only capture the cumulative behavior of R invocations of G_3, we cannot in general infer a mathematical relation between $C_{3,3}^F$ and $\sigma_{X^{G_3}}^2$. On the other hand, G_1 is always invoked exactly once per invocation of F, therefore $Y_1^F = X^{G_1}$ and $N^F = N^{G_1}$, allowing us to infer $C_{1,1}^F = \sigma_{X^{G_1}}^2$. For the same reasons as with G_3, the if-statement prevents us from expressing $C_{2,2}^F$ in terms of $\sigma_{X^{G_2}}^2$ for G_2.

2.1 Classifying Variance

We use Chebyshev's inequality [6] given below to determine if a function F's behavior can be considered *highly variant* over its N invocations. **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.*

$$Pr(|X - \bar{X}| \geq t\sigma) \leq \frac{1}{t^2} \tag{3}$$

For our purposes, the data samples refer to the N observations of the random variable X. 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 $\frac{1}{t^2} = 1 - 0.96 = 0.04$ and $t\sigma = 2 \times \bar{X}$. Therefore $\frac{\sigma}{\bar{X}} \geq 0.4$ becomes the condition for high-variance. Consequently we use the Coefficient-of-Variability metric for classifying the variant nature of nodes: $CoV = \frac{\sigma}{\bar{X}}$. In summary, nodes with $CoV \geq CoV_{Thresh}$ $(= 0.4)$ are labelled **high-variant**.

2.2 Underlying source of variance

Our interest in studying the variant behavior of a function F is to determine if other functions are the root underlying source of the variance in F. Consider Listing 2 that provides the implementations for functions G_1, G_3 and G_2 (invoked from Listing 1) along with additional functions invoked by G_2.

Listing 2: Functions invoked by F

```
void G1(...) {
  A1();
  SS; //local code statements
  A2();
}
void G3(...) {
  B1();
  SSS; //local code statements
  B2();
}
void G2(...) {
  if (...)
    D1();
  while (...)
    D2();
}
void D1(...) {
  P();
}
void D2(...) {
  Q();
}
```

Consider the following scenarios:

1. Perhaps variations in G_1's execution time is simply reflecting variations in A_1's execution time. Then G_1 is simply a *transmitter* of A_1's variance, that ultimately contributes to F's variance. A_1 would be then the *underlying origin* of F's variance even though F does not invoke it directly. Further analysis may reveal that A_1 itself is merely a transmitter of variance originating further down the call-chain.

2. Alternatively, the local code statements denoted by SS may be causing G_1's behavior to be variant, with A_1 and A_2 only playing a minor role. This would establish G_1 itself as an underlying origin of F's variance.

3. Another alternative is that none of SS, A_1 and A_2 have large variances of their own. But if their behavior varies in synchrony, then their limited variance gets amplified: $\sigma^2_{X^{G_1}} = C^{G_1}_{0,0} + C^{G_1}_{1,1} + C^{G_1}_{2,2} + 2*(C^{G_1}_{0,1} + C^{G_1}_{0,2} + C^{G_1}_{1,2})$, where each term in the sum is not large individually, but all are positive leading to a large $\sigma^2_{X^{G_1}}$. Again, this would establish G_1 as a root underlying source of F's variance.

In order to determine which of the above three scenarios apply, we use the fact that $C^F_{1,1} = \sigma^2_{X^{G_1}}$ (established in the preceding paragraphs), and directly compare the values of the $C^{G_1}_{i,j}$ terms against $\sigma^2_{X^F}$. This lets us determine if $\{A_1\}$, $\{A_2\}$ or $\{A_1, A_2\}$ could replace G_1 as a significant contributor of variance to F.

Unfortunately, the availability of $C^{G_3}_{i,j}$ terms does not permit a similar analysis to be possible for determining if B_1 or B_2 are the underlying sources of F's variance with G_3 merely being the transmitter. The primary impediment is that a relation between $C^F_{3,3}$ and $\sigma^2_{X^{G_3}}$ cannot be established in general. We can get around this limitation by treating the body of G_3 as *implicitly inlined* into F. Then B_1 and B_2 will directly get terms in the $\mathbf{C}^F$ matrix instead of terms for G_3. The effect of the SSS block of statements would be absorbed into the Y^F_0 random variable. In theory, we could recursively inline any of the G_i functions to any depth in order to determine the underlying sources of variance despite the presence of loop-bodies and if-statements around call-sites. However, each depth of inlining call-chains originating at G_i requires the corresponding $\mathbf{C}$ matrix to be reconstructed. Additionally, the size and cost of constructing $\mathbf{C}^F$ grows as a square of the number of leaf call-sites in F after inlining. For any given F, the large number of combinations (for each depth along multiple call-chains originating under F) and the growing cost of constructing $\mathbf{C}^F$ for each combination precludes an exhaustive examination of possibilities via inlining. A further complicating factor is that the data relevant for reconstructing $\mathbf{C}^F$ for any fixed choice of F is scattered over the entire length of the profile. The profile consists of time-stamped events indicating function-entry and exits. Any function F may be invoked multiple times throughout the entire duration of execution of the application, leading to the aforementioned scattering. Given that the length of the profile data typically runs into billions of profile events (corresponding to the application executing for a few minutes), the reconstruction of $\mathbf{C}^F$ for a new inlining depth essentially requires a fresh pass over the entire profile data, even though the cost of the pass is amortized over all functions F that need their $\mathbf{C}^F$ reconstructed.

Given that each additional pass over the profile data is an enormously time-consuming operation, we utilize additional mathematical properties and construct heuristics to guide and limit the implicit inlining of call-chains within any F. G_2 illustrates this approach, where we can check if F invokes G_2 with *very high probability* (say, $> 95\%$). If so, we can expect that $\sigma^2_{X^{G_2}} \approx C^F_{2,2}$. In other words, the very few instances of F that do not invoke G_2 are unlikely to significantly deviate $C^F_{2,2}$ from $\sigma^2_{X^{G_2}}$. Therefore, we can meaningfully compare the $C^{G_2}_{i,j}$ terms against $\sigma^2_{X^F}$ in order to determine if D_1 or D_2 functions are the underlying origins of variance for F. Such tests can be cascaded: if D_1 is invoked with very high probability in G_2 then we can compare $C^{D_1}_{i,j}$ terms against $\sigma^2_{X^F}$ to see if function P is the underlying cause of F's variance. This heuristic can sometimes also work for call-sites within loops. If G_2 invokes D_2 only once with high probability, then we can compare $C^{D_2}_{i,j}$ terms against $\sigma^2_{X^F}$ to see if Q is an underlying cause of F's variance.

Once the cascaded application of the heuristic has identified several low-level functions as *potential contributors* (say, B_1, P and Q), we can invoke a single pass over the profiling data to *ascertain* whether they are indeed significant contributors to F's variance, and to *quantify* their variance contribution. Note that this quantification can be done for A_1 and A_2 without needing an additional profiling pass since $C^F_{1,1} = \sigma^2_{X^{G_1}}$ holds mathematically, not just in approximation. Now, depending on the determination of which of B_1, P and Q are significant contributors, the cascading heuristic can be re-applied to their callees to check if even lower level functions are the underlying originators of F's variance. Again, another pass of the profile data may be needed to verify the hypotheses of the heuristic. Ultimately, by using the mathematical equivalence (where it applies) and the heuristic in many other cases, we can significantly reduce the number of profiling passes needed while not compromising on either the depth or validity of the contribution relationships discovered.

2.3 Implicit Inlining Notation

If F has been implicitly inlined along various call-chains to levels where its body now contains the call-sites for functions $A_1, A_2, ..., A_i, ...$, we use the following notation to refer to the correspondingly modified terms. Later sections will define metrics using this notation.

- Random variable $Y^{F|A_i}$ refers to the cumulative execution time spent in A_i during a given invocation of F. This is analogous to Y_i refering to the cumulative execution time of immediate call-site G_i.
- $\mathbf{C}^{F|A_1,A_2,\cdots}$ refers to $\mathbf{C}^F$ that has been modified to remove terms for the G_is that have been inlined away, and has new entries added for $A_1, A_2, ...$, i.e., the new $Y^{F|A_i}$ random variables get entries alongside the remaining Y_js.
- $\mathbf{C}^{F|A_1,A_2,\cdots}_{i,j}$ refers to terms between an unaffected Y_i and Y_j, i.e., it's the same as $\mathbf{C}^F_{i,j}$. However, $\mathbf{C}^{F|A_1,A_2,\cdots}_{A_i,j}$, $\mathbf{C}^{F|A_1,A_2,\cdots}_{i,A_j}$ and $\mathbf{C}^{F|A_1,A_2,\cdots}_{A_i,A_j}$ refer to new terms between $Y^{F|A_i}$ and Y_j, between Y_i and $Y^{F|A_j}$, and between $Y^{F|A_i}$ and $Y^{F|A_j}$, respectively.

3. CONTEXT SENSITIVITY OF BEHAVIOR

Characterizing the context sensitivity of behavior can significantly help with program understanding. In many parts of a typical program, behavior is determined largely by data or parameters passed from the calling context. A module in the program may repeatedly request a similar type of processing from a library, which may be distinct in terms of data-set size or characteristics from the processing requested by other modules. This can make the behavior exhibited by the library specific to the calling module, thereby making the library's behavior context-sensitive in that program.

A context aware analysis scheme would determine *whether* context affects behavior and *what aspects* of the context affect behavior. Context-sensitive analysis offers several benefits. Some examples are: specialization of functionality to context, either by the programmer or by the compiler in a manner akin to trace-driven optimizations; detecting behavioral bugs (such as real-time deadline violations) that predominantly show up in some contexts of invocation of function F and not in others.

In addition to the generic benefits outlined above, incorporating context is particularly important for studying variance in execution time. Variance is difficult to study purely in terms of the behavior of the lexical constructs in the application, such as summarizing variance over all invocations of a basic block or a function. Unlike total execution time, which is the driving metric in hot-spot analysis, *variance does not naturally accumulate hierarchically.*

In contrast, with variance, which lacks the property of hierarchical summarization, we need to *explicitly* examine the role that

calling-context plays in affecting lower-level behavior. The per-invocation variance in a lower-level function-call L does not directly add to the per-invocation variance of a function H higher up in the call-chain: L's variance may be neutralized by another function J inside H whose variations are negatively correlated with those of L (due to large negative cross-terms, as per Eq 2); or, L and J may contribute a much larger variance to H than the sum of their variances if they vary in synchrony (due to large positive cross-terms, as per Eq 2); or, H may invoke L conditionally or within a loop in a manner that altogether decouples the variance of L from the variance observed in H. Yet, it is important to study variance at all levels at which it occurs in the program. For example, in a packet routing application, it may be important to characterize the variance in both the per-packet routing time at each port, as well as in the overall packet routing rate for the router. Such an analysis would help designers/programmers determine how much buffer space should be provided at the port for transmission/reception and the amount of global memory needed for temporarily storing received packets. This can also facilitate a trade-off between the sizes of port buffer space and global memory. A video compression application would similarly benefit from a characterization of the variance in frame-compression times at the high level, and per-image-block variations in processing time within a video-frame for diagnosing which types of image-blocks or types of processing are responsible for the high level variance in frame-compression-time.

3.1 Summarizing Occurences of Variant Behavior using Call Structure

Figure 1 shows the dynamic call structure of the application under a function H. Each of the nodes represent potentially multiple invocations of the corresponding function. Nodes with identical names differ in terms of the call-chain used to invoke them under H. Such a program representation is referred to as the Call-Context-Tree (CCT) in literature [2]. Henceforth, let F_i be used to label the nodes for the same function F (in contrast with Section 2 where G_i referred to distinct function names). We will now use N^{F_i}, $\bar{X}^{F_i}$, $\sigma^2_{X^{F_i}}$ and CoV^{F_i} to refer to metrics on the corresponding *node instances* of F, and similarly for node instances of other functions. The ideas developed in Section 2 to study the variance behavior of functions now apply in a corresponding manner here to study the variance behavior of nodes.

In this example, the nodes annotated with greek letters represent the occurences of high variant behavior (as per the CoV test in Section 2.1). Whenever a function F has nodes exhibiting high variance (F_1, F_2, F_3 and F_4), it becomes important to *contrast* against the occurences of low-variance behavior of the same function F (here F_5, annotated with lv). Drawing such a contrast between high-variant and low-variant behavior, and between different kinds of high-variant behavior is crucial for infering *unique* circumstances under which each behavior occurs. Here, each greek letter identifies a distinct type of high-variant behavior. Multiple nodes may have similar variant behavior (F_1, F_2 and F_4 for behavior α). *Similarity* in variant behavior is determined by comparing the $\left(\bar{X}, CoV\right)$ tuples for the corresponding nodes. All occurences of low-variant behavior are annotated with lv. For example, L has high-variant behavior in L_1 and low-variant in L_2. Functions J, U, V and W exhibit high-variance in their only nodes. Functions that never exhibit high-variant behavior do not need their node instances distinguished for the purpose of variance analysis. Hence their nodes do not get subscript indices to distinguish between them (A, B, M, etc).

Figure 2 shows how the variant behavior under H would be summarized using *minimal distinguishing call chains* (MDCCs) introduced by Kumar, et al. [12, 11]. Their primary concern was to identify the shortest call-chain segments that allowed each behavior of a function F to be distinguished from other distinct behaviors. There is no need to distinguish between multiple occurences of the same behavior. In fact, it is highly desirable that very few, relatively short call-chain segments summarize numerous instances of the similar behavior in the CCT. For example, if Figure 1 represents the behavior observed during a *profiling run* of the application, then the following *predictions* would hold for any subsequent *regression run* of the application, provided we can reasonably assume that the profiling run behavior is a good representative of expected regression run behaviors. The invocations of function F whenever the top of the program call-stack is B will collectively exhibit the statistics (mean, CoV) for behavior α. In fact, this is predicted to apply collectively to invocations of F which have either B or $Q \rightarrow M \rightarrow C$ on the program call-stack (i.e., the call-chain suffixes identified by MDCCs(α)). Similarly statistics for β are collectively predicted for those invocations of F that had any sequence in MDCCs(β) occuring at top of stack (in this case, MDCCs(β) has only one sequence). Similar predictions hold for the lv behavior of F, and lv, η, λ, $\lambda^{/}$, etc behaviors of other functions. L needs to be distinguished based on the index of its lexical call-sites under H since the two call-sites produce different behavior. J has a null MDCCs(η) since every invocation of J would count towards behavior η.

While MDCCs may provide a concise representation that predicts the expected behavior of functions based on the current call-stack, several challenges from the point of view of facilitating *program understanding* become evident:

1. **How to preserve Structural Relationships?** Each function F, L, etc has their MDCCs constructed independently. Any relationships between the behaviors of a group of functions is obfuscated. For example, it is not evident from the corresponding MDCCs that F exhibits behavior α when invoked under L that exhibits behavior η, and F exhibits lv when invoked under L exhibiting lv.

2. **How to eliminate/minimize Redundant information?** Redundant information that does not contribute towards program understanding can quickly dilute the useful information present in the results. The majority of the space in the MDCCs results is occupied by long call-chain segments that do not themselves describe behavior, even though they distinguish between behaviors of F, etc. In fact, the behaviors of F are much more simply distinguished using L_1, L_2, K and J in this example, with the added advantage of distinguishing the behavior closer to its *cause* (i.e. knowing which of L_1, L_2, K or J is invoked will determine F's behavior). Further, the MDCCs constructed by Kumar et al. distinguish call-chains all the way from *main* as there is usually no good automated way to select an appropriate H deeper in the call-structure. This leads to potentially very long call-chains.

.

The overall challenge now is to have a generalized scheme for eliminating redundant information, preserving structural relationships where they matter, and still retaining the behavior distinguishing advantages of MDCCs. In proposing the Variance Characterization Graph (VCG) representation we take a different approach to meet this overall challenge. Figure 3 shows the VCG representation of the variant behavior under H. The behavior α of CCT nodes F_1 and F_2 still gets summarized into a single *VCG node*. The two lexical instances of L (on edges tagged [0] and [3] under H) along with J and the *absence* of K distinguish the spectrum of F's behavior. At the same time the variant behavior of J and L is also fully characterized. The *Call-Context-Set* (CCS) for each VCG node is shown. In contrast with MDCCs, a CCS only gives the call-chain paths to a VCG node from the VCG node's parent. Also, we choose to retain the full path segments in the CCS rather than attempt to find the shortest distinguishing segments. We do so because the call-segments originating from the immediate parent VCG node are usually very short, and therefore we retain their full length for improved clarity. Note that we can still apply MDCC analysis just on the CCSs if we need to predict the occurence of each child VCG node based on the program call-stack. Each VCG node has metrics tagged to it describing its behavior.

Let's make a number of observations about the VCG representation under H. First, only those functions that show variant behavior in atleast some instances under H show up as VCG nodes (let's refer to them as the *used functions*). Yet, there is little to no loss in the ability to *locate* every distinct behavior for each used function in the VCG tree. The only ambiguity in this example is between the VCG nodes for F under the two L's. For this the programmer can examine the corresponding CCSs associated with those two L's and determine the difference in the call-chains to each L. Therefore, location of behavior is implicitly disambiguated by the tree structure itself, with a need to examine CCSs only if a VCG node has more than one identically named direct children VCG nodes. More explicitly, K versus J distinguished F's α versus β behavior, so there is no

```
void H(){
  L(…);   //lexical site [0] L1
  K(…);   //lexical site [1]
  J(…);   //lexical site [2]
  for(… i<Num …)
    L(…); //lexical site [3] L2
  U(…);   //lexical site [4]
}

void L(…) {
  if (…) {
    A(…);    //lexical site [0]
    D(…);    //lexical site [1]
  }
  else N(…); //lexical site [2]
}
```

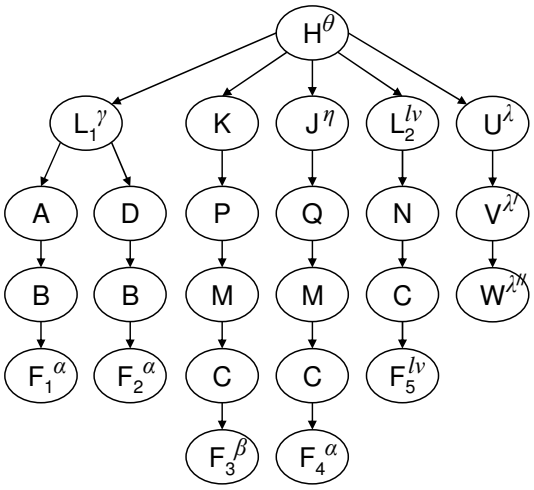


Figure 1: Example application behavior under invocations of function H. *High-variant node instances are annotated with greek letters. Instances of these functions with low variance are annotated 'lv'. Functions that are low-variant in all instances do not have their instances annotated or indexed.*

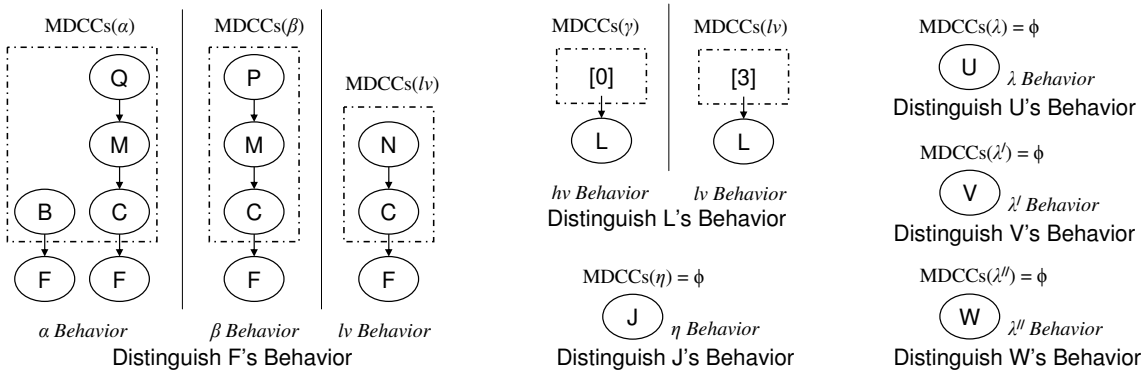
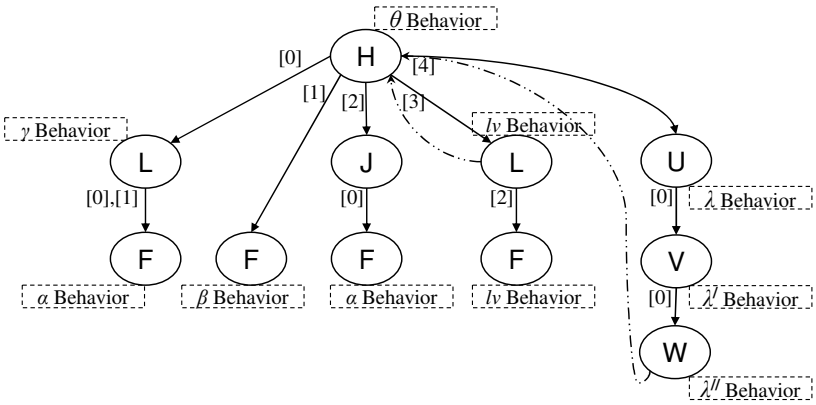


Figure 2: Minimal Distinguishing Call-Chain representation of variant behavior under H. *Emphasis is on summarizing identical behavior and unambiguously locating each behavior on call stack under H. Structural relationships are lost. Leads to numerous long MDCCs if $H = main$.*



VCG Node	*Call-Context-Set*
H[0]-L	{[0]}
H[0]-L[0][1]-F	{[0]-A[0]-B[0], [1]-D[0]-B[0]}
H[1]-F	{[1]-K[0]-P[0]-M[0]-C[0]}
H[2]-J	{[2]}
H[2]-J[0]-F	{[0]-Q[0]-M[0]-C[0]}
H[3]-L	{[3]}
H[3]-L[2]-F	{[0]-N[0]-C[0]}
H[4]-U	{[4]}
H[4]-U[0]-V	{[0]}
H[4]-U[0]-V[0]-W	{[0]}

Figure 3: Variance Characterization Graph and Call Context Sets representing variant behavior under H. *Emphasis is on summarizing structural relationships between behaviors. Call Context Sets use hierarchy to unambiguously locate behavior, and can often be ignored when structure itself provides adequate location information. Variance contribution relationships annotated on tree structure as dashed backedges, and allow skipping of redundant variance transmitter nodes (U, V).*

need for long MDCCs. Also, examine the CCT call-chain from H to F via K, appearing in the VCG representation as just $H \rightsquigarrow F$. Since K, P, M, C are all *unused functions*, and since we have to account for every CCT instance of used functions L, J, etc including low-variant ones for contrast, the programmer examining the call-structure of the application can eliminate any call-chains containing L, J, U, V, W as candidates for $H \rightsquigarrow F$. This leaves the call-chain starting with K as the only possibility. This was inferred without examining any CCSs. However, if there are multiple call-chains culminating in F but containing only unused functions, then CCSs would have to be referred to for disambiguating locations of distinct behavior.

Since VCG retains structure, we can now superimpose long-range variance contribution relationships over this structure. In this example, let's say that the analysis described in Section 2.2 determined the *lv* L and W as the underlying sources of variance for H. This creates $H \rightsquigarrow L$ and $H \rightsquigarrow U \rightsquigarrow V \rightsquigarrow W$ as *linear variance contributing segments*, or *linear segments* for short. The head of the linear segments, H in both cases here, is the *target* high-variant node to which variance is being contributed by the *end* nodes L and W of the linear segments, respectively. The nodes *internal* to the linear segment, U and V in the second linear segment, are merely the transmitters of underlying variance. As developed in Section 2.2, the low-variant L can also contribute variance to H if it is invoked inside a loop with a loop-count that varies over invocations of H. Strictly speaking, the variance is originating in the loop-block rather than in L, but we make a design decision to restrict our analysis and the VCG representation to just functions for tractibility. The dashed backedges from L to H and W to H graphically capture the *variance contribution relationships* corresponding to the two linear segments. The backedges serve to indicate the presence of linear segments and demarcate their spans. The two backedges are referred to as contributing variance to the [3] and [4] *variance components* of H, respectively. The [*i*] variance component refers to the i^{th} lexical call-site of H that has been found to be a significant contributor of variance (to σ^2_H) as per Section 2. Each variance component of H can, in general, have multiple backedges targeting it, corresponding to multiple underlying VCG nodes contributing significant variance to H through the same component of H. The overall structure shown in Figure 3 is referred to as a *VCG Pattern*.

4. DOMINANT BEHAVIOR

The CCT is constructed with *main* as the root node. Therefore, the analysis described in Section 3 will find a *forest of VCG Patterns* rather than a single pattern, as *main* itself is invoked only once and is therefore not variant. In our experiments, the number of patterns detected in the CCT could range from a few dozen to several hundred, depending on the application benchmark. Some patterns may have a VCG node F that exhibits unusually high variance (i.e., large CoV^F) or has an unusually large total execution time over all its invocations (called large *weight* W^F, where $W^F = N^F * \bar{X}^F$). Either one of these characteristics make this node *dominant*, and make it important that the containing pattern P be presented prominently in tha analysis results to the programmer. A single pattern may consist of nodes whose dominant characteristics vary by orders of magnitude. However, it is also possible that there are multiple patterns $P_1, P_2, ..., P_M$, with identical structure, and F is not dominant in any of them. But the behavior of F becomes dominant if all of these structurally identical patterns are considered together. In this case we are seeing a behavior that is imporant to capture and present to the programmer, not because it is dominant in a given instance, but because it is a *common recurring pattern of behavior*. Therefore, we need to be able to *distill the dominance* of recurrent behavior from across the application's CCT. We do this by merging identical patterns and creating a *higher level VCG pattern* that retains the same structure while accumulating the statistics of the individual patterns.

In general, the desire to summarize information into as few patterns as possible needs to be counterbalanced with a need to preserve the distinctiveness of behaviors. We refer to this as the *summarization vs precision* tradeoff. To allow for maximal summarization, we need to have the ability to merge patterns that are just similar rather than identical in structure and corresponding metrics. When considering two patterns for merging, differences can arise at multiple points between them: a subtree present in one may be absent in the second, the statistics associated with individual nodes may differ, the amount of variance contributed by corresponding linear segments may differ. We take a weighted sum of these differences (the number of terms in the sum grows with the size of the patterns), and construct a single scalar *merge-cost* which encompasses all these factors. Depending on the intensity of the *summarization pressure* to merge, the analysis may overlook differences between two patterns to a greater or lesser degree. The merge-cost is compared against a threshold derived from the summarization-pressure to determine if the given two patterns are merged. However, there are certain structural conditions that must be satisfied for a merger to be even considered. The conditions check whether the two patterns are structurally *compatible* (as opposed to identical) both in the tree structure that provides context, and in the contribution structure imposed on the tree structure (the linear segments, identified by the backedges). When the compatibility conditions are satisfied, the resulting merged tree has the structural characteristics of both lower level trees, and it meaningfully summarizes the behavior over the two trees, including over subtrees that are only present in one lower level pattern. The next section formally defines the VCG representation and describes how the annotated statistics are meaningful to the programmer for understanding behavior. The VCG representation allows a uniform interpretation of the statistics, independent of whether a given VCG pattern is a low-level pattern extracted from the CCT, or if it is the result of many steps of merging.

The summarization vs precision tradeoff is best served if the analysis achieves the maximum precision possible under any given setting of a **summarization-pressure parameter**, τ_S. In the service of this goal, we choose to use *hierarchical agglomerative clustering* [4] as the technique for merging patterns and distilling dominant behavior. Hierarchical agglomerative merging treats each pattern as a point. The distance between every pair of points is computed (we use the merge-cost as the distance measure). A pair of points a and b with the smallest distance $d_{a,b}$ are merged, provided $d_{a,b} \leq \tau_S$. This *consumes* points a and b and produces a new point m as a result of the merger. Other *unconsumed* points c and d can then be similarly merged in the order of smallest distance. This process consumes a *layer* of points and produces a new layer of points. The new layer consists of the unconsumed points from the previous layer and the merged points produced as a result of mergers in the previous layer. Additional layers are similarly produced, further reducing the number of points until no further reduction is possible. Every pair of points in the last layer is either incompatible (based on the compatibility conditions for merging two patterns) or the merge-cost exceeds τ_S.

The agglomerative merging process is first done recursively *inside* each pattern in bottom-up order. Essentially all subtrees under a VCG node are treated as separate points for the agglomerative merging process. This is exemplified by the merger of F_1 and F_2 under L_1 inside the H-rooted pattern in Figure 3. Under a sufficiently high summarization-pressure, the identical subtrees of the two L nodes for the η and *lv* behaviors would also have been merged, producing only one subtree for L under H, which is $L \rightsquigarrow F$. The resulting behavior for the merged F would still be α, but the merged L would get an altogether new behavior, δ, that would be *mathematically equivalent* to the cumulative behavior of all invocations of function L under H (since only one L remains under H). Of course, if there are multiple VCG patterns that have H as root, then the δ behavior of L occurs only under those invocations of H that correspond to this given pattern. The different patterns rooted in H are differentiated from each other by the CCS of H, i.e., the CCSs of the various Hs consist of *disjoint sets* of call-chain segments from *main* to H (i.e., a given call-chain segment can exist in the CCS of only one H). Now suppose that a variance contribution backedge also existed from the F under the L with η to the [0] component of H. This backedge would now prevent the merger of the two Ls because it will violate a *compatibility condition*: if the merger of the identical subtrees into a single $L \rightsquigarrow F$ subtree were allowed, then the resulting subtree would have backedges to H from both L and F, indicating that both L and F are the *root underlying contributors* of variance to H. However, L contains F, so F being a root underlying contributor contradicts L being a root underlying contributor. The compatibility conditions and the justification of the mathematical equivalence alluded to above will be covered in the next two sections.

Once recursive internal merger has been attempted on all pat-

terns collected from the CCT, these patterns form the initial layer on which agglomerative merging will be attempted across patterns. The patterns in the final layer produced by the agglomerative merging process are *prioritized* and *trimmed*. Prioritization sorts the patterns in the order of their dominance. The dominance of a pattern is computed as a combination of the root node's dominance and the recursive dominance of the subtrees. Prioritization is also done recursively among the subtrees in each pattern. Prioritization, both within and across patterns, provides the most important results to the programmer first. The next step of trimming can optionally be performed across patterns and recursively within each pattern. The goal of trimming is to eliminate entire patterns, or subtrees within patterns, which are substantially less important than the more important ones. We use the *Variance Impact Metric* introduced by Kumar, et al. [11] ($VIM \triangleq \sigma N$ per node, summed over pattern) to drive both prioritization and trimming. A **trim-threshold parameter** $0.0 \leq \beta < 1.0$ eliminates patterns (and, subpatterns at any level within a pattern) whose VIM is less than β times the VIM of the highest-VIM pattern (i.e., most dominant) at that level.

Prioritization and trimming help achieve *generality* with the analysis results. If only the most re-inforced and weighty behavior is presented to the programmer, then chances are good that this behavior derived over a *profiling data-set* will also be the most dominant behavior that occurs over any future runs of the application on *regression data-sets*. Strictly speaking the exact same metric values are unlikely to appear, however, we can expect that essentially the same pattern structures, in essentially the same order of relative dominance will be found in the regression runs. On the other hand, disabling or minimizing trimming will allow *data-set specific behavior* to become apparent. When comparing two sets of untrimmed results for data-sets with different characteristics, we can expect that there will be significant structural and ordering differences in the less dominant parts of the results. These expectations are borne out by our experimental validation on real world applications.

5. VCG PATTERN

Here we describe the properties of a VCG pattern. These properties and interpretations apply equally well to both the unmerged patterns extracted from the CCT, as well as patterns resulting from extensive internal and cross-pattern merging.

Tree Structure and Contribution Structure.

A VCG pattern consists of a tree structure with back-edges. The nodes represent functions invoked in the dynamic call structure of the application. A forward-edge establishes *context*, indicating that a child node represents a function-call invoked during the execution of the function-call represented by the parent node. The child function-call may have been invoked indirectly by the parent via a long chain of intermediate calls. A backedge establishes *variance contribution*, indicating that the source node is the underlying cause of the execution-time variance observed in the target node. The target node will be an ancestor of the source node in the tree structure. The backedges collectively annotate a variance contribution structure on the tree structure.

Linear Segment.

Each backedge delimits a linear segment, which is a sequence of VCG nodes. The first node in the linear segment is the *originator*, i.e., the target node for the backedge that is the head node for the linear segment in the tree structure. The last node is the *underlier* which contributes variance to the originator. There can be zero or more *internal nodes* between the first and last node in the sequence. We shall henceforth overload the term linear segment to refer to a data structure consisting of a sequence of pointers to VCG nodes.

Those functions that do not exhibit variant behavior in any context of invocation in the Call Context Tree representation are altogether excluded from appearing as VCG nodes. On the other hand, both the low-variant and high-variant contexts of execution of functions that are high-variant in atleast some CCT contexts must be retained in the structure of a VCG pattern.

A node F within a pattern P has the following attributes:

1. **Function name**: $F.fname$
2. **Node type**: $F.node\_type$. Each VCG node can be one of three types: `Task`, `Contributor` or `Contrast`. A `Task` is a node at which high-variant behavior is *principally observed*, rather than just being a transmitter of variance to a higher-level `Task`. A `Task` F may or may not have a backedge to another `Task` J which is F's ancestor in P. A `Contributor` is a low-variant node that is nonetheless the underlying source of variance to an ancestor `Task` J via a backedge. A `Contrast` node F is either low-variant (but there exists a high-variant VCG Node G elsewhere, such that $F.fname = G.fname$), or, F is transmitter of variance from some lower level `Task` or `Contributor` in P to a `Task` ancestor of F.
3. **Statistical Metrics**: $N^F, \bar{X}^F, \sigma^2_{X^F}, W^F = N^F * \bar{X}^F$. Here W^F is weight, i.e. total execution time spent in node across all its N^F invocations.
4. **Containing Linear Segment**: $F.cls$. If this node is contained in a linear segment, but is not the originating node, then points to the corresponding linear segment data structure. Otherwise, undefined ($\perp$).
5. **Originating Linear Segments**: $F.list\_ols$. A list of the linear segments that originate at this node, potentially empty. Each list element points to a unique linear segment data structure. That is, if the i^{th} linear segment originating at F is $[F, A, B, C, U]$, then $F.list\_ols[i] = A.cls = B.cls = C.cls = U.cls = same[F, A, B, C, U]$. Note that $F.cls$ will refer to a separate linear segment (if any) that started at some ancestor of F.
6. **Parent**: $F.parent$. The parent of F in the tree structure P. Undefined ($\perp$) if F is the root of P.
7. **Children VCG nodes**: $F.children[0..m]$. Pointers to the VCG nodes that are the immediate children of F in the tree structure of P. These children *do not* directly correspond to the lexical call-sites in the body of function $F.fname$. Instead, they could refer to the lexical call-sites out of order, could skip some lexical call-sites, or any single index could refer to multiple lexical call-sites at once (due to mergers).
8. **Call Context Set**: $F.ccs$. A set of call-chain segments giving the one or more paths in the CCT that locate the one or more CCT nodes corresponding to F from the one or more CCT nodes corresponding to $F.parent$. If $F.parent = \perp$ (i.e. at root of P), then gives the paths from *main*. A call-chain segment consists of a sequence of tuples of the form $(fname, [i])$, where $fname$ is a function-name and $[i]$ refers to the i^{th} lexical call-site in the body of $fname$. The first tuple of the sequence always has an empty $fname$. This is illustrated in Figure 3.
9. **Originating Linear Segment Statistics**: $F.ols\_stats$ a list of statistics associated with backedges incident on F. The entries correspond to the linear segments in $F.list\_ols$.

6. MERGING PATTERNS

A pattern P_i refers to the *root node* of the pattern, from which the rest of the pattern can be traversed. Two patterns P_1 and P_2 are merged using a three pass algorithm. The first pass, recursive Function MergePatternTrees, produces a *merge-candidate* VCG tree structure P_M, assuming no incompatibility is found between the tree structures of P_1 and P_2. Each node in P_M has a pointer to the corresponding node in P_1 and/or P_2. The corresponding node may possibly be absent in atmost one of P_1 or P_2. Therefore, the tree structure produced in P_M is an *overlap* of the tree structures of P_1 and P_2.

The first stage of the function, lines 2 - 4, checks if P_1 and P_2 match on the name and types of their root nodes. The second stage, lines 5 - 21, performs a **call-context compatibility check** over the call-contexts from the P_1 and P_2 to their children. Note that this is the mechanism that determines which child of P_1 corresponds to which child of P_2 (if any). The ordering $P_1.children[0..m_1]$ and ordering $P_2.children[0..m_2]$ *do not establish a correspondence* between the children of P_1 and P_2. The idea is that if a newly created child node *merge_child* of P_M has a call-context *cc* assigned to (i.e., $cc \in merge\_child.ccs$), and *cc* is in the call-context of some child, *child*, of P_1, then this must imply that *i) child* under P_1 becomes the corresponding node for *merge_child* under P_M (and similarly for P_2) , and *ii)* every other call-context $cc_1 \in child.ccs$ must necessarily also be mapped to *merge_child*. The second requirement may fail to be satisfied if some $cc_1 \in child.ccs$ has already been assigned to a different merge-child under P_M. The call-context compatibility check verifies the satisfaction of the second requirement, and

causes the merge process to fail if it is violated. The intent here is that the call-contexts of *child* must not get split over multiple children of P_M. Therefore, $child.ccs \subseteq merge\_child.ccs$, as $merge\_child.ccs$ is being created to accomodate the call-chains of corresponding nodes $child_1$ and $child_2$ from P_1 and P_2, and $child_1.ccs$ may not be identical to $child_2.ccs$ due to internal merging of subtrees within a pattern. The last stage, lines 22 - 36, recursively invokes Function MergePatternTrees on pairs of children drawn over P_1 and P_2 that have been found to be in corresponce (based on call-contexts). This creates *mcount* merged children subtrees under P_M.

Function MergePatternTrees(P_1, P_2)

```
   input  : VCG Patterns P1 and P2 to merge, atmost one of which can be ⊥
   output : Merged VCG Pattern PM if tree structure merge successful, else ⊥
1  begin
2      if P1 ≠ ⊥ ∧ P2 ≠ ⊥ then
3          if P1.fname ≠ P2.fname ∨ P1.node_type ≠ P2.node_type then
4              return ⊥
5      all_P1_ccs ← Union of all P1.children[i].ccs if P1 ≠ ⊥, else φ
6      all_P2_ccs ← Union of all P2.children[i].ccs if P2 ≠ ⊥, else φ
7      all_ccs ← all_P1_ccs ∪ all_P2_ccs
8      Initialize Assigned[cc] ← ⊥, ∀cc ∈ all_ccs
9      mcount ← 0
10     foreach cc ∈ all_ccs do
11         if Assigned[cc] = ⊥ then
12             Assigned[cc] ← mcount
13             mcount++
14             if P1 ≠ ⊥ ∧ ∃i s.t. cc ∈ P1.children[i].ccs then
15                 if ∃cc1 ∈ P1.children[i].ccs s.t. Assigned[cc1] ≠ ⊥ ∧ Assigned[cc1] ≠ Assigned[cc] then
16                     return ⊥
17                 Assigned[cc1] ← Assigned[cc], ∀cc1 ∈ P1.children[i].ccs
18             if P2 ≠ ⊥ ∧ ∃i s.t. cc ∈ P2.children[i].ccs then
19                 if ∃cc2 ∈ P2.children[i].ccs s.t. Assigned[cc2] ≠ ⊥ ∧ Assigned[cc2] ≠ Assigned[cc] then
20                     return ⊥
21                 Assigned[cc2] ← Assigned[cc], ∀cc2 ∈ P2.children[i].ccs
22     PM ← Create VCG Node with shared/only fname and node_type attributes
23     PM.CorrNodes ← [P1, P2]
24     for 0 ≤ m < mcount do
25         combined_child_ccs ← {cc : Assigned[cc] = m}
26         child_P1 ← ⊥
27         if P1 ≠ ⊥ ∧ ∃i s.t. Assigned[cc] = m where cc ∈ P1.children[i].ccs then
28             child_P1 ← P1.children[i]
29         child_P2 ← ⊥
30         if P2 ≠ ⊥ ∧ ∃i s.t. Assigned[cc] = m where cc ∈ P2.children[i].ccs then
31             child_P2 ← P2.children[i]
32         merge_child ← MergePatternTrees(child_P1, child_P2)
33         if merge_child = ⊥ then return ⊥
34         merge_child.ccs ← combined_child_ccs
35         merge_child.parent ← PM
36         PM.children[m] ← merge_child
37     return PM
```

The second pass, Function MergeContributionStructure, takes the merged tree structure P_M (with its intrinsic pointers to nodes within P_1 and P_2) and returns it with the corresponding contribution structures also merged, provided no incompatibility exists between the contribution structures of P_1 and P_2.

Line 3 is the **originating linear segment compability check**. This checks if a node *cnode_i* in, say P_1, is originating some linear segment, LS, which does not have a corresponding linear segment originating at *cnode_j*, the node in P_2 that corresponds to *cnode_i* in P_1 via P_M. However, if $cnode_j$ is missing the child subtree that could have carried the linear segment corresponding to LS (as tested by the existence under $cnode_j$ of a node corresponding to the second node of LS), then no conflict in interpretation is introduced by allowing the contribution structure to merge. Lines 5 - 7 are the **linear segment internal compatibility check**. This verifies if a node $cnode_i$, say in P_1, is internal to a linear segment LS, and the corresponding node $cnode_j$ in P_2 is internal to a linear segment identical to LS. However, if corresponding node $cnode_j$ does not exist (i.e., $cnode_j = \perp$), then no conflict in interpretation of the merged structure is introduced at *mnode*.

The third pass updates the metrics on the merged nodes

Function MergeContributionStructure(P_M)

```
   input  : Merge Candidate VCG Pattern PM, with merge nodes pointing to corresponding nodes in P1 and P2
   output : PM with contribution structure merged if compatible, else ⊥
1  begin
2      foreach mnode in pre-order traversal of PM tree do
3          if ∃cnode_i, cnode_j ∈ mnode.CorrNodes, cnode_i ≠ cnode_j ∧ cnode_i, cnode_j ≠ ⊥ s.t. ∃LS_i ∈ cnode_i.list_ols with no corresponding LS_j ∈ cnode_j.list_ols, and, cnode_j has a child node corresponding to the second node in the LS_i sequence then
4              return ⊥
5          if ∃cnode_i, cnode_j ∈ mnode.CorrNodes, cnode_i ≠ cnode_j ∧ cnode_i, cnode_j ≠ ⊥ s.t. cnode_i.cls ≠ ⊥ then
6              if cnode_j.cls = ⊥∨ Node sequences cnode_i.cls and cnode_j.cls are not in correspondence via PM then
7                  return ⊥
8          foreach cnode_i ∈ mnode.CorrNodes do
9              foreach LS_i ∈ cnode.list_ols do
10                 merged_LS ← Construct a sequence of nodes in PM that corresponds to LS_i (in P1 or P2 as case may be)
11                 if merged_LS ∉ mnode.list_ols then
12                     Append merged_LS to mnode.list_ols
13                     foreach mn in merged_LS node sequence except first node (i.e., mnode) do
14                         mn.cls ← merged_LS
15     return PM
```

as described in the next subsection. Then it computes the **Kolmogorov-Smirnov difference measure** $0.0 \leq D \leq 1.0$ between the original metrics m_1 and m_2 coming from P_1 and P_2, respectively. This KS measure [6] is a standard statistical technique for studying the difference between two probability distributions. We treat the m_1 and m_2 metrics as Gaussians for the purpose of computing D. If the node or edge corresponding to, say the m_2 metric, does not exist in P_2 (since we allow the merging of dissimilar patterns), an appropriately zeroed out value is used for m_2 in computing D. This would make a merger of dissimilar structures automatically more expensive since a zero-distribution will have a relatively high D difference from non-trivial distributions. The algorithm then computes a weighted average of the D values over the merged tree. The W_i measures (total execution time) of merged tree nodes are used to weigh the associated D in the average. Since we want a strong dissimilarity at the root node ($D > \tau_S$) to by itself rule the merger to be prohibitively costly, we choose to use a bread-first-traversal to *progressively* add nodes (and corresponding D terms) to the weighted average. If the progressive average exceeds the summarization pressure τ_S at any point, we prohibit the merger. Otherwise, a scalar merge-cost d_{P_1,P_2} is produced. Therefore, the programmer can choose τ_S in the range 0 to 1, where a low τ_S only allows highly similar structures and behaviors to merge thereby providing *precision*, whereas a larger τ_S would *force the summarization* of results into much fewer, more compressed patterns that will possibly *generalize* more readily to future runs of the application.

If a merge-cost $d_{P1,P2}$ is produced, then $((P_1, P_2), d_{P1,P2}, P_M)$ provides a pair of points for the agglomerative clustering algorithm to consider. Agglomerative clustering will use $d_{P1,P2}$ as the corresponding distance measure, and will use P_M as the result of combining P_1 and P_2.

6.1 Merging Statistics

Consider corresponding nodes F_1 and F_2 from P_1 and P_2 respectively, that get merged into F_M under P_M. The root nodes P_1 and P_2 (since a pattern is just a root to the pattern tree) have had their call-contexts from *main* combined under P_M, i.e., $P_M.ccs = P_1.ccs \cup P_2.ccs$.

Under P_i there is a path of VCG nodes to F_i, with call-context sets at each level on this path. Now, $F_1.fname = F_2.fname = F_M.fname$ since the merger was successful. Let us refer to that common function name as $fname$ for brevity here. Therefore, every *full call-chain* (all the way from *main*) that results in invocations of $fname$ that got counted in N^{F_1}, can be enumerated as some concatenation of the call-chain-segments taken from the call-context-sets along the VCG path to F_1. Note that the converse is not necessarily true: every full call-chain that can possibly be constructed by taking segments from the call-context-sets in a VCG path may not have actually occured during the profil-

ing execution of the application. But we don't need the converse property to hold.

Now, once the patterns P_1 and P_2 are merged, we would like the same property to hold for F_M under P_M. Clearly, this *enumerable path property* still holds as the call-context-set at each merged node is a superset of the corresponding nodes in P_1 and P_2. Therefore, in terms of annotated statistics, we need to still combine the N^{F_1} invocations of $fname$ for F_1 and the N^{F_2} invocations for F_2 into N^{F^M} invocations for F_M. The programmer should be able to interpret the statistics uniformly for P_1, P_2 and P_M without needing to know if some of these patterns are the result of merging one or more layers of lower level patterns. This can indeed be done as follows: $N^{F_M} \leftarrow N^{F_1} + N^{F_2}$, $\bar{X}^{F_M} \leftarrow \frac{\bar{X}^{F_1} * N^{F_1} + \bar{X}^{F_2} * N^{F_2}}{N^{F_M}}$, $\sigma^2_{X^{F_M}} \leftarrow \frac{N^{F_1} * (\sigma^2_{X^{F_1}} + (\bar{X}^{F_1} - \bar{X}^{F_M})^2) + N^{F_2} * (\sigma^2_{X^{F_2}} + (\bar{X}^{F_2} - \bar{X}^{F_M})^2)}{N^{F_M}}$. These can be verified as being *mathematical equivalences*. That is, even though we no longer have the N^{F_1} separate observations of r.v. X^{F_1} and the N^{F_2} observations of r.v. X^{F_2} available, we can still exactly compute the resulting metrics as if we had re-computed the mean, variance directly from the combined observations. The secondary metrics, CoV^{F_M} and W^{F_M}, can ofcourse always be recomputed from the primary metrics of mean, variance and count.

For a `Task` node H with incoming backedge b_i corresponding to the i^{th} linear segment originating at H, i.e., $H.list\_ols[i]$, the *variance contribution* statistics are recorded as a tuple in $H.ols\_stats[i] = T_i$. Let b_i represent the contribution of a lower level node W. Then, $T_i = (\bar{Y}^{H|W}, \sigma^2_{Y^{H|W}}, N^H)$ is a tuple which can be updated during mergers just like the node statistics above. The *contribution fraction* along b_i is a secondary metric $cf_i \leftarrow \mathbf{C}^{H|W}_{W,W}/\sigma^2_H$ (notation: see Section 2.3). The numerator term is updated on merger like the variance terms above.

7. DOMINANT VARIANCE ANALYSIS

Dominant Variance Analysis consists of the following steps:

Step 1 **Profiling and CCT Construction**: Ammons, et al. [2] proposed the Call-Context-Tree representation as a compact representation (compared to a Dynamic Call Graph) for profiling context-sensitive behavior. They proposed low-overhead compiler-instrumentation techniques and the use of hardware counters to minimize perturbations in measurements. Zhuang, et al. [20] construct *approximate* CCTs to further lower profiling overhead while retaining most of the analysis coverage of a full CCT. Their work is also an example of instrumenting the Java runtime instead of the application code. Our analysis framework works with any profiling and CCT construction methodology. The additional annotations needed by our analysis (mean, variance, etc) are straightforward to incorporate in any CCT construction scheme. For this paper, we chose to profile-instrument C applications using the LLVM compiler infrastructure [13]. Once the CCT is constructed, high-variant nodes are identified based on the CoV test in Section 2.1.

Step 2 **Identifying long-range underlying sources of variance on CCT**: Section 2.2 describes how additional profiling passes over the profile sequence allow the identification of low-level functions that significantly contribute to variance of functions several levels up in the CCT.

Step 3 **Initial Extraction of VCG patterns**: Once the annotated CCT has been constructed and long-range relationships identified, certain CCT nodes get labelled as `Task`, `Contributor` and `Contrast` based on the criteria in Section 5 for *Node Type*. For example, nodes H, L_1, F_1, F_2, F_3, J, F_4, U, V and W from the CCT in Figure 1 get labelled `Task`; node L_2 gets labelled `Contributor`, and node F_5 gets labelled `Contrast`. As illustrated, `Task` nodes exhibit high-variance behavior, possibly contributing significant variance to a higher-level `Task` (W to H). A `Contributor` is not high-variant but it contributes significant variance (say due to enclosing loops) to a higher-level `Task` (L_2 to H). `Contrast` nodes do not exhibit or contribute variance, but are important to contrast against since other node-instances of the same function are `Task`s or `Contributor`s. *VCG trees* consisting only of these labelled nodes are extracted top-down from the CCT. The *unused* CCT nodes (such as A, B and D in Figure 1) may become part of the Call Context Sets of relevant VCG nodes. Therefore, a single VCG edge may traverse several levels of unused nodes in the CCT (Figure 3). These *initially extracted* VCG trees form the *base-layer of VCG patterns* on which further merging will be attempted subsequently.

Step 4 **Merging within and across VCG patterns**: The algorithms in Section 6 attempt merging *internally* within each base-layer VCG pattern, then perform merging *across* the subsequently produced patterns (Section 4).

Step 5 **Prioritization and Trimming**: Using the VIM metric, the final merged layer of VCG patterns are sorted in priority-order both internally (i.e., recursively within a pattern) and across patterns, so that the most dominant results are presented first to the programmer. The β parameter sets the trimming threshold to eliminate non-dominant results (Section 4).

8. EXPERIMENTAL EVALUATION

The emerging importance of interactive media applications on consumer desktops, embedded systems and surveillance was a key motivation towards our developing the Dominant Variance Analysis. These applications are complex compositions of multiple modules (possibly from several developers). The function-level control flow corresponds closely to the distribution of application functionality exercised by a given data-set. The corresponding spread of execution-time and its variability not only helps reveal the general design of the application and the data-set specificity of functionality, but is also crucial for characterizing the real-time responsiveness likely to be percetable to an interactive user (i.e., QoS). Therefore, we chose video benchmarks from MediaBench II [7] for validating our technique.

We use the following benchmarks, two video encoders and two decoders: `mpeg2enc`, `mpeg2dec`, `h263enc` and `h263dec`. Each of these benchmarks consists of between 90–120 functions. We ran all profile generation experiments and the Dominant Variance Analysis on an Intel Q6600 system (quad-core) clocked at 2.40GHz with 2GB of RAM. The input data-sets used for profiling were reference videos (for decoders) and corresponding decoded raw-image sequences (for encoders). The function-timing information in the profile was collected by running the applications within the Simics architectural-simulator on an x86-based system image. We used *cycle-counts* of the simulated CPU to time-stamp function entry and exit events.

`dolbyrain`, `hockey1_cif` and `baikonur_r7_overflight` are standard videos used to characterize video codecs. `qos` is a trailer of the *Quantum of Solace* action movie transcoded to two resolutions (320x240 and 640x480), since pixel-count dramatically affects execution-times [7]. We used between 100–300 frames of the video sequences for profile generation.

All the steps (Section 7) of the Dominant Variance Analysis were implemented in python. Step 1 took between 1400–5800 seconds (over different benchmarks and data-sets). Step 2 took between 6000–16500 seconds. Steps 3–5 cumulatively consumed less than 1 second since they operated only on the constructed CCT and not on the profile sequence. We expect Steps 1 and 2 to be substantially faster if implemented in C/C++.

Table 1 shows the sizes of the representations at various stages of the analysis, with different settings of the summarization-pressure parameter τ_S (over its full range: 0.0–1.0) and with trimming enabled and disabled. $\beta = 0.1$ performs a relatively non-aggressive trimming (Section 4), only eliminating patterns/sub-patterns at a level that are atleast an order to magnitude ($10\times$) less dominant than the most dominant pattern/sub-pattern at that level. The constructed CCT is a tree rooted at *main* with approx. 300–900 nodes (across different benchmarks and data-sets). The initial layer of VCG patterns extracted from the CCT range from 75 nodes in 3 patterns to 407 nodes in 184 patterns. Now consider the final merged VCG patterns produced. With *trimming disabled* (columns under $\beta = 0$), only summarization pressure can force a reduction in number of VCG nodes and patterns. The achievable reduction is limited by the various **compatibility conditions** (Section 6) which must be satisfied for merger to be possible, regardless of summarization pressure. $\tau_S = 0.01$ represents *very low summarization pressure* (recall, overall merge-cost of two patterns is the *average* of all the D difference measures between corresponding nodes/backedges, and that $0.0 \leq D \leq 1.0$). Since $\tau_S = 0.01$ shows substantial reduction in number of nodes and patterns for all benchmarks, this suggests that the initial extracted layer of VCG patterns had a large number of patterns that matched closely in structure and in corresponding annotated statistics (hence, behavior). This re-

Benchmark	Data-set	CCT size	Initially Extracted (unmerged)	No trimming ($\beta = 0$) τ_S 0.01	0.2	0.4	0.6	0.8	1.0	With trimming (using $\beta = 0.1$) τ_S 0.01	0.2	0.4	0.6	0.8	1.0
mpeg2enc	qos320x240	707	78/6	42/5	31/5	25/5	25/5	25/5	25/5	32/1	23/1	17/1	17/1	17/1	17/1
	qos640x480	695	74/3	39/3	28/3	21/3	21/3	21/3	21/3	31/1	22/1	17/1	17/1	17/1	17/1
	dolbyrain	718	75/3	30/3	29/3	23/3	23/3	23/3	23/3	23/1	23/1	17/1	17/1	17/1	17/1
mpeg2dec	qos320x240	890	400/186	42/11	35/8	35/8	35/8	35/8	35/8	15/4	16/5	16/5	16/5	16/5	16/5
	qos640x480	920	407/184	55/12	49/10	49/10	49/10	49/10	49/10	16/4	17/5	17/5	17/5	17/5	17/5
	dolbyrain	898	394/181	42/7	41/6	41/6	41/6	41/6	41/6	14/2	15/3	15/3	15/3	15/3	15/3
h263enc	hockey1_cif	747	207/71	50/7	44/5	44/5	44/5	44/5	44/5	6/3	2/1	2/1	2/1	2/1	2/1
	baikonur_r7	745	45/12	27/9	18/6	18/6	18/6	18/6	18/6	8/4	2/1	2/1	2/1	2/1	2/1
h263dec	hockey1_cif	294	125/52	40/9	35/9	35/9	35/9	35/9	35/9	8/2	8/2	8/2	8/2	8/2	8/2
	baikonur_r7	296	134/55	51/13	43/13	35/12	35/12	35/12	35/12	22/4	23/4	15/3	15/3	15/3	15/3

Table 1: #-of-nodes/#-of-patterns: initially extracted, versus, after merging on τ_S without/with trimming

sult demonstrates that the CCT representation had *significant recurrent patterns of behavior* that were merged by our analysis to *distill the dominance of the associated recurrent behavior*. Under larger τ_S relatively limited additional merger occurs, producing *stable* VCG results in most cases with $\tau_S \geq 0.2$, and in all cases with $\tau_S \geq 0.4$.

The trends discussed with trimming disabled also hold true when trimming is enabled (table columns under $\beta = 0.1$). In addition to the results *stabilizing* with $\tau_S \geq 0.2$ for most cases, and $\tau \geq 0.4$ for all cases, we observe a *convergence* of the results across different data-sets: the number of VCG nodes and patterns becomes almost identical under any fixed setting of $\tau_S \geq 0.2$ (discrepancy for `h263dec` is explained later). Therefore, we see stability moving across the rows and convergence moving along columns for a given benchmark. The final VCG patterns are output to a text file. We use a Mathematica script to parse the textual results and visualize them. We find that under stability the resulting VCG graphs are *identical in structure and statistics*. This is expected since all results in a row of the table are constructed from the same profile sequence, just summarized or trimmed to different extents. On the other hand, under convergence the *structure of the VCG graphs* is *virtually identical*, differing perhaps with the addition/deletion of one or two nodes and edges. The annotated statistics differ substantially over different data-sets, but they still produce similar *contribution fractions* on corresponding backedges. This is a more surprising result since the data-sets used (and hence the profiles produced) were quite different for results in the same column. The combined effect of stability and convergence is that under low trimming, all results for a benchmark with $\tau \geq 0.4$ are essentially identical in structure and contribution-backedge-structure, thereby revealing an application's ***inherent*** **variant-behavior structure**.

8.1 Usage Model

The programmer needs to run Steps 1–2 only once for a data-set of their application to generate and save a CCT. Steps 3-5 (taking $<$ 1 sec) can then be repeatedly invoked exploring τ_S and β parameters. Initially, $\tau_S = 0.4$ with trimming enabled should produce a concise VCG representation, enablng the programmer to understand which handful of functions (out of possibly hundreds making up the application code) are relevant for understanding variant behavior. Next, a small $\tau_S < 0.1$ would more precisely distinguish between structure and behavior and expose the calling-context-sensitivity of these differences. Next, reducing/disabling trimming would allow the *data-set specific* differences in behavior to become apparent between VCGs from two different data-sets. We illustrate this process using VCGs generated for our benchmarks.

Figure 4 shows the VCG for `mpeg2enc`, consisting of 17 nodes in a single pattern. Boxes with solid outlines present `Task` nodes, ovals present `Contrast` nodes. To ease reference, a unique id is attached to the function-name of a node. For example, the root node is *motion_estimation* with id **(0)**. Further, the VCG reveals what is well-known in literature [7, 10] for `mpeg2enc`: motion-estimation is by far the most significant cause of per-frame variability in execution-time, and this variability occurs due to searching for matching image-blocks in adjacent video-frames ($fullsearch$ nodes). The $dist1$ function computes a distance measure between offset image-blocks, performing half-pixel interpolation if needed, and accounts for most of the variance of the parent $fullsearch$ (seen as contribution backedges). Examining `mpeg2enc`'s source code reveals that $frame\_ME$ has multiple call-sites for $frame\_estimate$, and usually conditionally invokes any one call-site per invocation of $frame\_ME$. This explains $frame\_estimate$ **(2)** contributing 129% (i.e, $>$ 100%) of the variance in $frame\_ME$ **(1)**, since the different call-sites of $frame\_estimate$ also contribute large negative cross-correlation terms (Eq 2). The μ, γ, N and V terms give the node's mean, CoV, invocation-count and VIM metric, respectively. The $frame\_estimate$ function contains multiple call-sites to $fullsearch$. For lower τ_S fewer of these call-sites get merged, leading to the larger number of nodes seen in Table 1. Multiple less dominant pattern-trees (*dct_type_estimation*, *predict* for macro-block form prediction) get eliminated by enabling trimming. Subtrees $frame\_estimate$ **(2)** and **(6)** do not get merged despite matching structure and high τ_S because **(2)** has a contribution-edge to parent but **(6)** does not (i.e., incompatibility in contribution-structure). Note, $dist1$ **(5)** and **(9)** have $\gamma \geq CoV_{Thresh} = 0.4$ despite being `Contrast` nodes, due to merger of multiple low-variance nodes under high-pressure.

A programmer can characterize the primary soft real-time QoS of `mpeg2enc` by looking at the per-frame encoding times of *motion_estimation* **(0)**. For the `qos320x240` data-set, we get $\mu = 493.9 \times 10^6$ cycles and $\gamma = 0.518$. Assuming a 1GHz CPU clock, this implies a mean encoding time of 0.493 seconds/frame with $<$ 4% probability (Section 2.1) that a frame-time will exceed $(1+\frac{\gamma}{0.2}) \times \mu = 1.77$ seconds (or, a $<$ 25% probability that a frame-time will exceed $(1+\frac{\gamma}{0.5}) \times \mu = 1.003$ seconds). If the desired mean encoding frame-rate is, say $1second/frame$, then the programmer may want to increase `mpeg2enc`'s *search-window-size parameters* [7] in order to allow wider searches for matching blocks (producing higher quality video or more compression), at the cost of greater likelihood of missing frame deadlines. We re-profiled the `qos320x240` data-set, this time doubling all the search-window-size parameters. This produced: $\mu' = 722.1 \times 10^6$ cycles and $\gamma' = 0.584$. This translates to a mean encoding time of .722 seconds with $<$ 4% chance of a frame-time exceeding 2.83 seconds (or, a $<$ 25% probability that a frame-time will exceed 1.565 seconds).

Figure 5 shows `mpeg2dec` with 15 nodes in 4 patterns, 3 of the patterns being trivial but sufficiently dominant avoid being trimmed compared to *decode_macroblock* **(0)**. The patterns and subpatterns are sorted in descending VIM, and given ids in depth-first order. The per-frame decoding function and the two main types of decoding functionality, intra and non-intra, get clearly highlighted. Additionally, frame-data-I/O functions are seen as potentially a significant factor in causing variability. Programmers consulting the reference design of the application may have otherwise missed the significance of frame-data-I/O in this case.

Figure 6 and Table 1 show the extremely dominant variability of $MotionEstimation$ in `h263enc`, and $SAD\_Macroblock$ within it, leading to all other functions being trimmed out. For low τ_S, the 3–4 separate instances of the $MotionEstimation \rightarrow SAD\_Macroblock$ pattern stay unmerged due to significantly differing statistics.

Figure 7 (for `h263dec`) mainly shows pattern trees for *reconstruct* **(0)**, the per-frame reconstruction function, and *getblock* **(6)** which performs frame-data-I/O. *recv* **(2)** and $rec4$ **(3)** are examples of `Contributor` nodes (dashed outlines), which have low CoV but contribute variance (and large negative cross-correlation

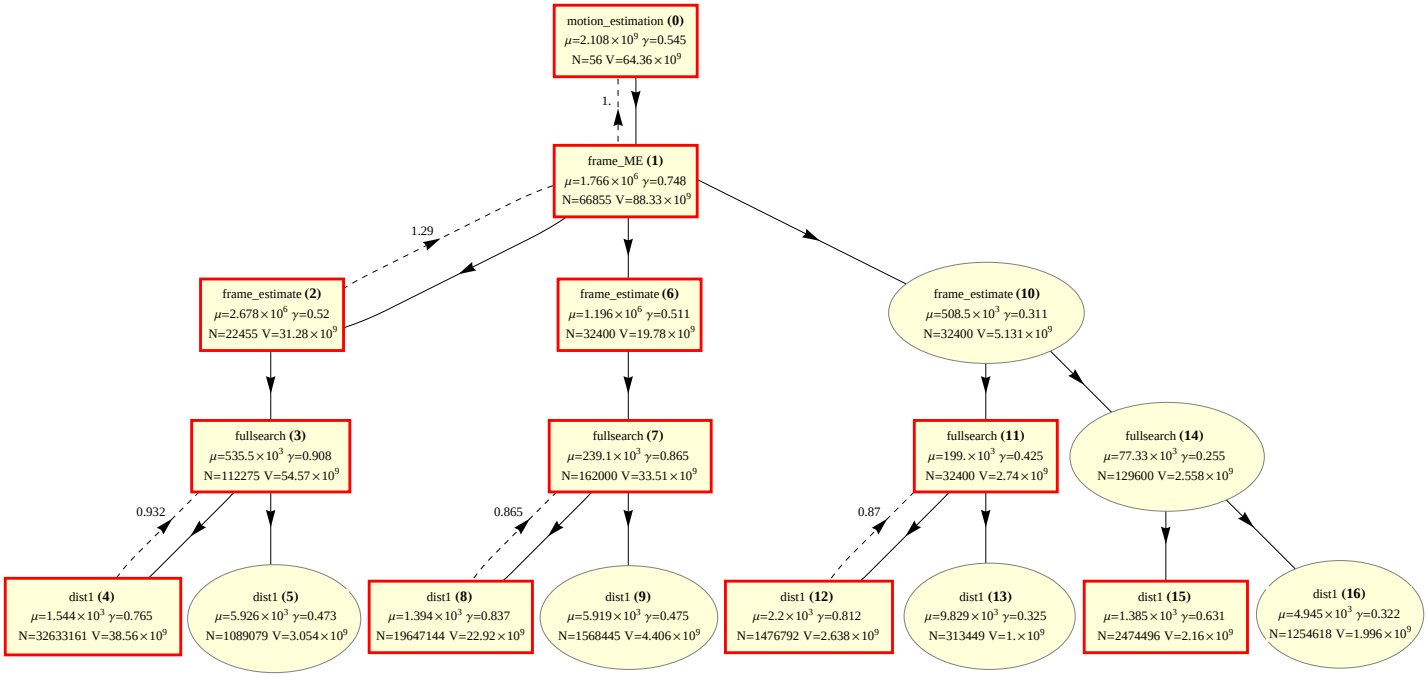


Figure 4: VCG for `mpeg2enc` with: `qos640x480`, $\tau_S = 0.4$, $\beta = 0.1$

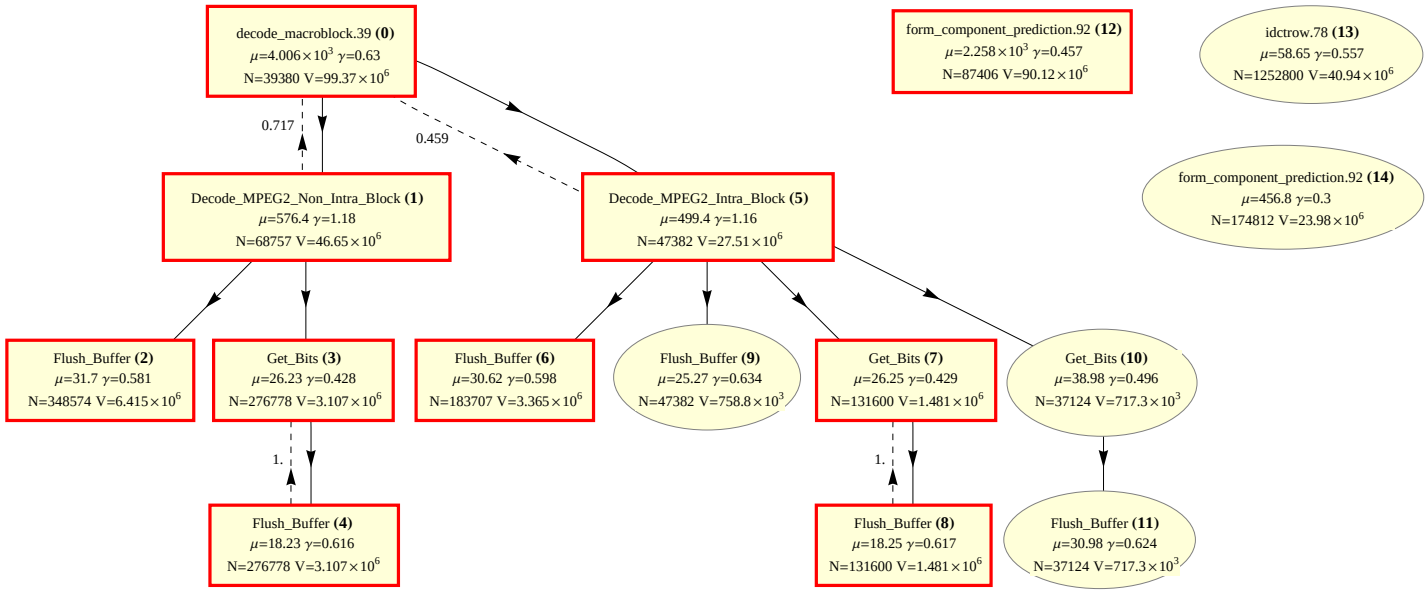


Figure 5: VCG for `mpeg2dec` with: `qos320x240`, $\tau_S = 0.01$, $\beta = 0.1$

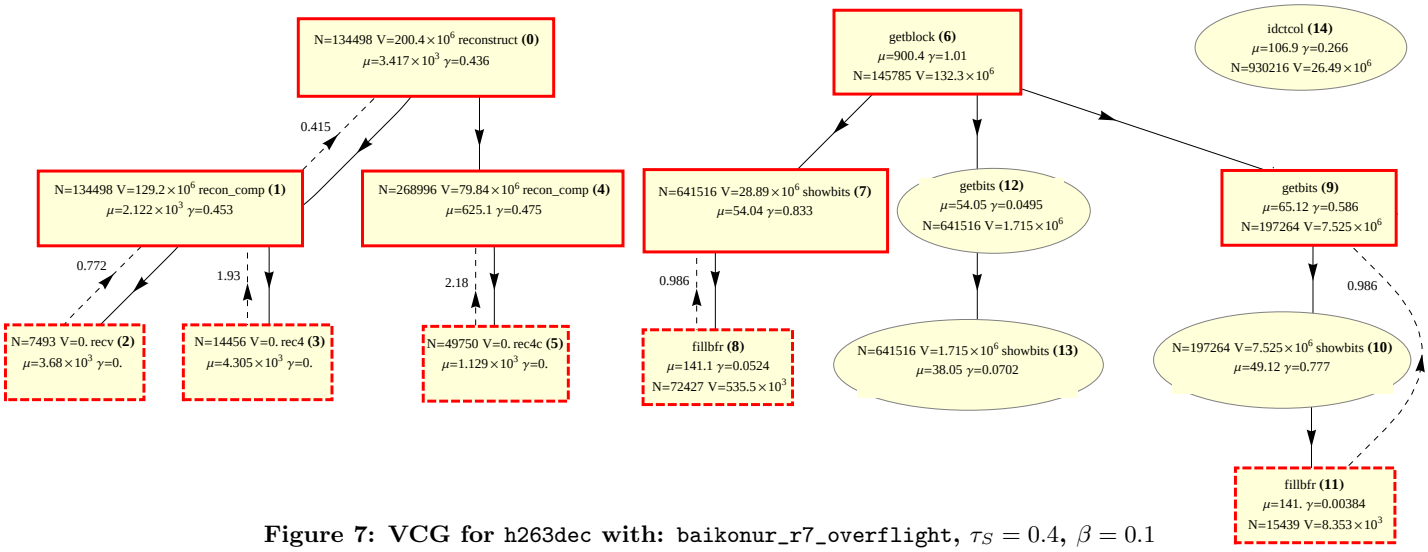


Figure 7: VCG for `h263dec` with: `baikonur_r7_overflight`, $\tau_S = 0.4$, $\beta = 0.1$

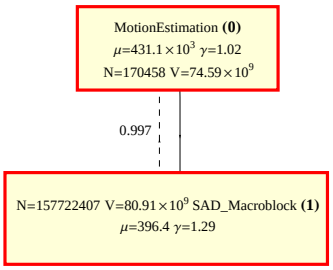


Figure 6: VCG for `h263enc`: `hockey1_cif` $\tau_S = 0.2$ $\beta = 0.1$

terms) to *recon_comp* **(1)** due to mutually exclusive conditional invocation. Also, *recon_comp* **(1)** and **(4)** subtrees seem to exhibit very different behaviors. The corresponding call-sites can be disambiguated based on the Call-Context-Sets (dumped separately). Here **(1)** and **(4)** correspond to lexical call-sites {3} and {4,5}, respectively, in the body of *reconstruct*.

9. RELATED WORK

Existing application profiling techniques look for program hot-spots and hot-paths [5, 9, 3]. These techniques attempt to find performance bottlenecks in an application, and do not attempt to characterize patterns of variant behavior.

Calder et al. have used statistical techniques to characterize large scale program behavior using few recurrent intervals of code [18] and to find phase change points in the dynamic execution of a program [14]. However, their work does not attempt to characterize the variant behavior in terms of the *functional decomposition* of the application. In particular, they seek out intervals in [18] with closely matching set of dynamic basic-blocks which cannot directly be related to the behavior of functions, and nor are sensitivities to the call-context detected.

Variability Characterization Curves (VCCs) and Approximate VCCs [16] 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 [17] 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 [15, 19] 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) [8] is an analysis methodology applicable to monolithic applications, and has been incorporated into commercial products such as from AbsInt [1]. 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.

As illustrated in Section 2, our work differs significantly from prior work [12, 11] on identifying variant behavior and call-context sensitivity. Rather than just identify short call-chain segments that predict the occurence of distinct behavior in individual functions, we are able to reconstruct the *variant structure* of the application using the VCG representation we introduce, providing not only a more intuitive interpretation of results, but also a more precise description of the call-context sensitivity.

10. CONCLUSION

In this paper we have illustrated the value of profiling the variant behavior of an application. We introduced the VCG representation to readily allow the application wide variant behavior of the application to be succintly represented. We introduce the Dominant Variance Analysis framework that allows precision of results vs summarization of results tradeoffs to be readily made by tweaking a single parameter τ_S. The VCG representation captures the variant structure of the application that we expect corresponds closely to the high-level design of the application. Therefore, our analysis aids program understanding by *i)* allowing a programmer unfamiliar with the application to identify the most important parts of a complex application, and *ii)* empowering even knowledgeable programmers with a succint, application wide characterization of the variant and context sensitive nature of the behavior to guide them in program optimization.

Future Work.

Our framework does not actually rely on the r.v. X representing execution time. Any other property of the program or architecture that accumulates hierarchically (say, the number of cache-misses in a function) can be substituted for execution time in X. Without needing any modifications, our proposed analysis and the VCG representation lends itself to the study of variance and context-sensitivity of that property. We seek to study the variant behavior of a number of such properties in the future.

11. REFERENCES

[1] http://www.absint.com.
[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97*.
[3] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. In *OOPSLA '02*.
[4] W. Day and H. Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *Journal of Classification*, 1(1):7–24, December 1984.
[5] E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35(11):202–211, 2000.
[6] J. E. Freund and R. E. Walpole. *Mathematical statistics (4th ed.)*.
[7] J. E. Fritts, F. W. Steiling, J. A. Tucek, and W. Wolf. Mediabench ii video: Expediting the next generation of video systems research. *Microprocess. Microsyst.*, 33(4):301–318, 2009.
[8] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS '06*.
[9] R. J. Hall. Call path profiling. In *ICSE '92*, pages 296–306, 1992.
[10] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. *SIGARCH Comput. Archit. News*, 29(2):254–265, 2001.
[11] T. Kumar, R. Cledat, J. Sreeram, and S. Pande. Statistically analyzing execution variance for soft real-time applications. *LCPC 2008*, pages 124–140, 2008.
[12] T. Kumar, J. Sreeram, R. Cledat, and S. Pande. A profile-driven statistical analysis framework for the design optimization of soft real-time applications. In *ESEC-FSE '07*, 2007.
[13] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO'04*, Mar 2004.
[14] J. Lau, E. Perelman, and B. Calder. Selecting software phase markers with code structure analysis. In *CGO '06*.
[15] C. Lin and S. A. Brandt. Improving soft real-time performance through better slack reclaiming. In *RTSS '05*, pages 410–421, 2005.
[16] Y. Liu, S. Chakraborty, and W. T. Ooi. Approximate vccs: a new characterization of multimedia workloads for system-level mpsoc design. In *DAC '05*, pages 248–253.
[17] M. Roitzsch and M. Pohlack. Principles for the prediction of video decoding times applied to mpeg-1/2 and mpeg-4 part 2 video. In *RTSS '06*, pages 271–280, 2006.
[18] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. *SIGOPS Oper. Syst. Rev.*, 36(5):45–57, 2002.
[19] E. Wandele and L. Thiele. Abstracting functionality for modular performance analysis of hard real-time systems. In *ASP-DAC '05*.
[20] X. Zhuang, M. J. Serrano, H. W. Cain, and J.-D. Choi. Accurate, efficient, and adaptive calling context profiling. In *PLDI '06*, pages 263–271.