Exploiting Inter-Sequence Correlations for Program Behavior Prediction

Bo Wu*, Zhijia Zhao*, Xipeng Shen*, Yulian Jiang†, Yaoqing Gao‡, Raul Silvera‡
*The College of William and Mary, Williamsburg, VA, USA
†Google, USA
‡IBM Toronto Lab, Toronto, Canada
{bwu,zzhao,xshen}@cs.wm.edu, yunlian@gmail.com, {ygao,rauls}@ca.ibm.com

Abstract

Prediction of program dynamic behaviors is fundamental to program optimizations, resource management, and architecture reconfigurations. Most existing predictors are based on locality of program behaviors, subject to some inherent limitations. In this paper, we revisit the design philosophy and systematically explore a second source of clues: statistical correlations between the behavior sequences of different program entities. Concentrated on loops, we examine the correlations' existence, strength, and values in enhancing the design of program behavior predictors. We create the first taxonomy of program behavior sequence patterns. We develop a new form of predictors, named *sequence predictors*, to effectively translate the correlations into large-scope, proactive predictions of program behavior sequences. We demonstrate the usefulness of the prediction in dynamic version selection and loop importance estimation, showing 19% average speedup on a number of real-world utility applications. By taking scope and timing of behavior prediction as the first-order design objectives, the new approach overcomes limitations of existing program behavior predictors, opening up many new opportunities for runtime optimizations at various layers of computing.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—optimization,compilers

General Terms Languages, Performance

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

Keywords Program behavior prediction, Dynamic optimizations, Dynamic version selection, Just-In-Time Compilation, Sequence prediction, Correlation

1. Introduction

Effective prediction of program dynamic behaviors—such as, loop trip-counts, function calling frequencies, dynamic dependences—has been long considered fundamental for program optimizations, resource management, and architecture reconfigurations (e.g., voltage scaling.) Its importance is underlined by the advents of multicore and cloud computing, which critically rely on behavior prediction for adapting to their complex process interplays, resource provision, and power management. Many kinds of behavior predictors have been implemented, in both hardware and software. Examples include last value predictors [3, 10, 20], stride predictors [10, 22], finite context predictors [11, 22], parameter stride [15], and memorization [21].

Despite design differences, all these existing predictors are essentially based on a common philosophy: to exploit the locality of the target behavior. A last value predictor, for instance, uses the return value of the previous invocation of a function as the predicted value of the current invocation [20]—what it exploits is the value locality (i.e., adjacent invocations share the same return values) of the target function. A stride predictor, on the other hand, exploits pattern locality by assuming that the return value changes by the same amount as it does in the previous invocation of the function [22]. Locality exploitation has been the essence of almost all runtime predictors.

The key observation in this work is that the locality-centric philosophy is powerful but incomplete. We use dynamic code version selection to illustrate its limitations. In this type of optimization, the runtime optimizer, at each invocation of a function, predicts which of several versions of the function that have different optimizations applied will work the best in this upcoming invocation [8]. A locality-based predictor, for its reliance on recent invocations of the

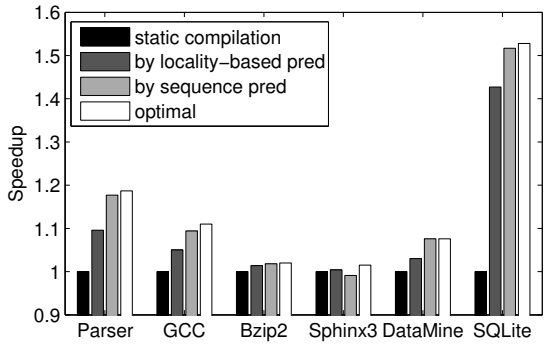


Figure 1. Speedups brought by dynamic version selection (details in Section 3.1.)

function as the clue for its prediction, faces two challenging scenarios. Figure 2 (a) illustrates the first. The function *cluster* in the program consumes more than 20% of the total execution time of a data mining program *DataMine*, but is invoked only once in each run. For lack of recent invocations, locality-based predictors are inapplicable[1]. The second scenario is when locality is missing, illustrated in Figure 2 (b). The graph shows the sequence of the best versions for 20 invocations of function *prepare_to_parse()* in one run of an English parsing tool *Parser* in SPEC CPU benchmark suites [14]. The randomness causes locality-based predictors to fail—the highest prediction accuracy is 33% in all locality-based predictors we have tried (detailed in Section 3.1.) The limitations throttle the power of dynamic optimizations significantly, as shown by the gaps between the "locality-based" and "optimal" speedup bars of six utility applications in Figure 1.

At a deeper level, the two examples reflect two inherent limiting properties that the locality-centric philosophy brings to existing predictors. The first is **reactivity**: Locality-based predictors are reactive to recent invocations of a function and hence fails in the first scenario, where no prior invocations exist. The second is **a small prediction scope**: Locality often weakens when the scope of prediction increases, because the more code the scope covers, the larger the behavior variety usually is. At the scope of the function in the *Parser* example, the locality is so weak that the predictors fail to work.

In this study, we advocate a new paradigm for predictor construction, the principle of which is to use inter-sequence correlations to complement locality to address the limitations of existing predictors.

Inter-sequence correlations are defined as statistical correlations among behavior sequences of different program constructs. A *program construct* here refers to a program component, such as a loop, function, statement. The behaviors of two constructs have a strong inter-sequence correlation if their sequences show some statistical correlations—that is, the sequences of one construct vary in a way not expected on the basis of chance alone but related with the sequences of the other construct. A simple example is that loop L_2 always iterates more or less quadratically as much as loop L_9 does. While such correlations sometimes arise from data flows, they are of a *statistical* property beyond data flows. Some may not correspond to a data flow at all and some may appear nondeterministic. But nonetheless, they hold statistically (i.e., with a large probability) across instances of the constructs, as well as across executions of the program.

Figure 3 shows an example. The shown "while" and "for" loops reside in two separate functions in *Parser*, which try to find respectively an underscore and a decimal point from a string. Both functions are invoked many times on different strings in one run of the program. The two graphs show the trip-counts or numbers of iterations of the two loops in an execution. Visually, we see large similarity between the graphs. Quantitatively, in most cases, the two loops have the same trip-counts in their corresponding invocations, despite that they are invoked far from each other and no clear data flows exist between them. Manual analysis uncovers the reason for the correlation: Most input strings have no underscores, while at the same time decimal points often show up only at the end of a sentence (as periods.) So the trip-counts of both loops tend to equal the string length.

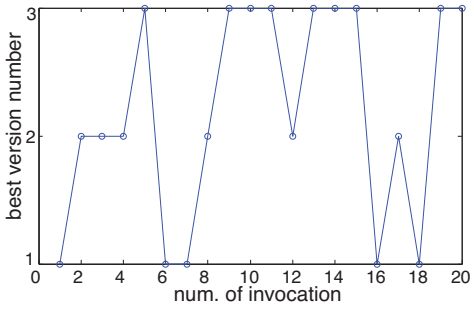
Correlations have been observed before (e.g., [16]). An example is the use of correlations among branches for branch prediction [27]. The target of such analysis and prediction has been individual instances (e.g., which branch the next instance of a conditional instruction will take), rather than behavior sequences (e.g., what the trip-count sequence of a loop will look like in a run.) Focusing on behavior sequences gives large scope and proactivity for behavior prediction, critical for overcoming the limitations of prior locality-centric predictors. It distinguishes the goal of this current work from those of prior studies on behavior correlations.

That focus is also the source of many new challenges. As illustrated by the trip-count sequences in Figure 3, a behavior sequence typically consists of either categorical or numerical behaviors manifested by hundreds or millions of consecutive instances of a program construct. Because different instances may be subject to different contexts, instances in one sequence may show dramatically different behavior values. Generally, when the target of consideration moves from a single instance to a sequence, significant complexity and pattern variety arise, as illustrated by the sequences in Figure 3 and more sequences in Figure 5. Many questions follow, including whether inter-sequence correlations exist broadly in a program, how to deal with intra- and inter-sequence behavior variations, how to detect inter-sequence correlations, and how to use them for large-scope behavior prediction and dynamic optimizations.

[1] The discussion here concentrates on runtime predictors, which are more broadly adopted than the alternative, offline profiling-based predictors due to input-sensitivity as Section 3.2 will show.

```
... ...
// clustering
CG = cluster();
for each group g of CG
  for each customer c in g
    if  c.subTime>12 then
      loadActivity (c);
...  ...
// processing
... ...
for each group g of CG
  process (bUseGPU, g);
... ...
```

(a)

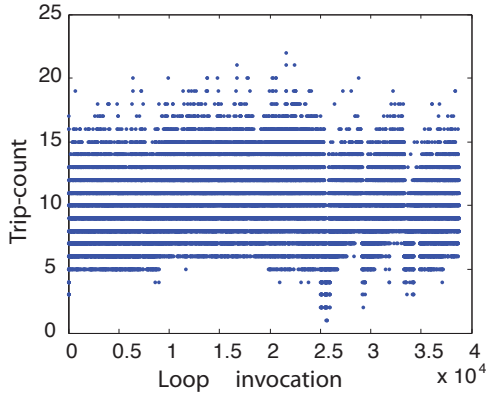


(b)

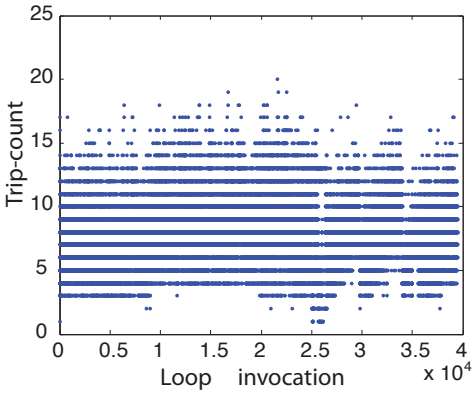
Figure 2. (a) Skeleton of a data mining program, *DataMine*, with an important function *cluster()* invoked once only. (b) Best versions of function *prepare_to_parse()* in *Parser*.

```
int contains_underbar(char * s) {
  /*  finding an underscore */
Loop 1:  while(*s != '\0') {
    if (*s == '_') return TRUE;
    s++;
  }}
```

```
int numberfy(char * s){
  /* finding a decimal point */
Loop 2:  for (; (*s != '\0') && (*s != '.'); s++)
    ...
}
```



(a) loop 1 trip-count sequence



(b) loop 2 trip-count sequence

Figure 3. Trip-count sequences of two loops in *Parser*.

This paper describes our investigation into these open problems. We first examine the existence of inter-sequence correlations in large scopes by measuring hundreds of program executions, billions of invocations to thousands of loops, and a large variety of patterns and code span. The measurement reveals not only the broad existence of correlations among program behavior sequences, but also produces the first taxonomy of program behavior sequence patterns, and uncovers some important properties of them.

We then analyze the complexities for detecting inter-sequence correlations and capitalization of them for program behavior prediction and optimizations. The investigation leads to a new type of runtime behavior prediction, named *behavior sequence prediction*, which, by complementing locality with correlations, addresses the limitations of existing predictors in both prediction timing and scope. For the problem in Figure 2 (a), for instance, it detects that a loop in the *loadData* function correlates with some major loops in function *cluster* in the *DataMine* program; the correlation makes it possible to replace the reactivity of prior predictions with **proactivity**, predicting the behavior of *cluster* from *loadData* even before *cluster* is ever invoked in this execution. For the *Parser* in Figure 2 (b), it finds that a loop in function *maxcost_of_sentence()* correlates with the loops in function *prepare_to_parse()*; using the correlation, it is able to overcome the randomness **large-scoped** behaviors often exhibit and predict the best versions of prepare_to_parse() at most of its invocations. As shown in the third bar group of Figure 1, the speedup almost doubles for a majority of the programs, equal or close to the optimal.

In summary, this work makes four-fold major contributions:

- It unveils the broad existence and some important properties of inter-sequence correlations.
- It creates the first taxonomy of program behavior sequence patterns.
- It proposes a new approach, *behavior sequence prediction*, that systematically exploits inter-sequence correlations for large-scoped program behavior prediction.
- By taking scope and proactivity as the first-order design objectives, the new approach overcomes limitations of existing predictors. It yields 19% speedup through function version selection and opens up many new opportunities for runtime optimizations at various layers of computing.

In the rest of this paper, we first examine the existence of inter-construct correlations and introduce a sequence predictor to translate the correlations into predicted program behaviors (Secton 2.) We then explores the uses in program optimizations (Section 3.) After a review on related work, we conclude the paper with a short summary.

2. Inter-Sequence Correlations and Sequence Predictor

In this section, we examine how broadly inter-sequence correlations exist and how they can be captured and formulated. Along the way, we develop a sequence predictor to systematically translate the correlations into predicted program behaviors. The predictor is used for assessing the strength of correlations, but has a much broader usage as Section 3 will show.

2.1 Methodology for Data Collection

Our measurement focuses on loops, the most important constructs in many programs. We add a pass in LLVM [19] to do instrumentation. The current framework works for C programs only. What we collect are the loop trip-count sequences of all C programs in SPEC CPU2006. Our measurement covers more than two hundred runs of eleven C programs with billions of invocations of thousands of loops. The program, *lbm*, is not included because all but one of its loops have constant trip-counts. We did not include *perlbench* because we have difficulties in making extra inputs work for it. Four programs in the benchmark suite, *gromacs, cactusADM, calculix, wrf*, are claimed as C programs, but contain Fortran components, and are hence not supported. We additionally include a SPEC CPU2000 program, *parser*, because this study started with it. Table 1 lists all used programs, along with the numbers of non-trivial loops—that is, those that have more than five instances in any run, and whose trip-count is not a constant. The machine we use is equipped with Intel Xeon E5310 processors, running Linux 2.6.22.

Table 1. Benchmarks and pattern distributions

Prog.	time var.	# of loops	Pattern Distribution					
			C1	C2.1	C2.2	C3.1	C3.2	C3.3
milc	40x	124	0.72	0	0.25	0.02	0.01	0
gobmk	9x	590	0.02	0.02	0	0.41	0.5	0.05
hmmer	39x	68	0.64	0	0.14	0.09	0.05	0.08
sjeng	58x	92	0	0.07	0	0.14	0.43	0.36
h264ref	81x	371	0.84	0.02	0.01	0.05	0.06	0.02
sphinx3	4x	299	0.59	0.02	0.1	0.08	0.04	0.17
bzip2	16x	174	0.5	0	0.03	0.26	0.09	0.12
gcc	11x	1884	0.14	0.03	0	0.64	0.08	0.11
libqu-antum	117x	40	0.63	0.04	0	0.25	0	0.08
mcf	25x	37	0.13	0.06	0.06	0.19	0.06	0.5
parser	8x	399	0.1	0.04	0.02	0.16	0.24	0.44

As a statistical approach, the sequence prediction requires a number of runs on different inputs per program to observe statistic properties. We hence collect some extra inputs based on our understanding of the typical usage of the program, learned through reading the source code, document, and default inputs. We finally have 20 inputs for each program, which stimulate different behaviors. The second column in Table 1 reports the ratios between the running times of the longest and shortest executions of the programs, reflecting the large differences in the input set.

2.2 Observations on Behavior Sequences

For understanding inter-sequence correlations, the first step is to understand behavior sequences themselves. This subsection reports four important properties of behavior sequences we observe from the 78,340 trip-count sequences we collected. Among these properties, two form obstacles for sequence prediction, while the other two reveal some opportunities.

Intra-Run Value Variation Due to the influence of invocation contexts, most sequences exhibit certain value variations. Of the 78,340 sequences, only 34% have a single value throughout the sequence. On average, the maximum trip-count in a sequence is 3620 times larger than the minimum.

Inter-Run Value Variation Most loops exhibit variations in the sequence values and lengths across different runs of the program. Of all 4078 loops, only 4% are counter-examples. On average, across the 20 runs of a program, a loop shows 6710% changes in the sequence length and 764% changes in the sequence mean value.

These two types of variations complicate the analysis of statistical correlations among the sequences, and form barriers for sequence prediction. But we also observe two favorable features of the sequences.

Behavior Sequence Taxonomy After manually examining a large number of sequences, we observe that although the sequences vary much, many of them fall into one of a handful categories in terms of the overall shape or pattern. It

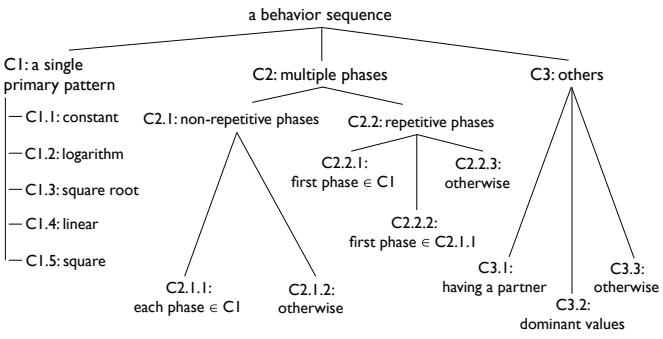


Figure 4. The taxonomy of program behavior sequences.

sheds an insight for using a divide-and-conquer strategy to tackle the sequence complexities: If we can categorize sequences into a handful classes, we may be able to design customized abstracts and solutions for each category.

After going through the sequences, we create a hierarchical taxonomy of program behavior sequences, as shown in Figure 4 and illustrated in Figure 5.

The design of the taxonomy is based on the complexity of the sequences (with predictability in mind.) There are three top classes in the taxonomy. A sequence belongs to class $C1$ if it can be modeled with a single primary pattern. The type of the primary pattern then puts the sequence into $C1.x$, where x can be 1, 2, 3, 4, 5, corresponding to the five types of primary patterns we concern in this work: constant, logarithm, square root, linear, and square relations with the occurrence number of a behavior instance. These five kinds of relations cover most of the patterns we have observed. Figure 5 (a) exemplifies a loop trip-count sequence of a linear pattern. For a sequence in $C1$, predicted values of the parameters of its primary pattern are enough for predicting the whole sequence.

A sequence belongs to class $C2$ if the sequence consists of multiple phases. Depending on whether the phases are repetitive, $C2$ is further divided into $C2.1$ and $C2.2$.

For a sequence with non-repetitive phases ($C2.1$), if every phase of it belongs to some class in $C1$, the sequence is essentially a combination of multiple $C1$ sub-sequences and can be hence represented by a series of parameters of the corresponding primary patterns. Such sequences form the subclass $C2.1.1$, exemplified by Figure 5 (b). Other sequences in $C2.1$ constitute $C2.1.2$, exemplified by Figure 5 (c). The modeling and prediction of $C2.1.2$ are similar to $C3$, discussed later in this section.

A sequence with repetitive phases ($C2.2$) can be easily predicted once its values in the first phase are known. Based on the patterns of its first phase, it can be further classified into three sub-classes. It falls into $C2.2.1$ if its first phase belongs to $C1$, exemplified by Figure 5 (d). In this case, that phase can be modeled as a sequence in $C1$. The sequence falls into the second sub-class, $C2.2.2$, if its first phase can be modeled as a sequence in $C2.1.1$—that is, its first phase itself contains multiple non-repetitive sub-phases and each of them is in a primary pattern. The first phase can be then modeled as a $C2.1.1$ sequence. Figure 5 (e) shows an example of $C2.2.2$. The $C2.2.3$ class consists of all $C2.2$ sequences that belong to neither $C2.2.1$.

The class $C3$ subsumes all sequences that cannot fit into either $C1$ or $C2$. These sequences usually show extreme irregularity. They cannot be modeled with a primary pattern, a combination of patterns, or phases. However, for some of them, their corresponding program constructs happen to have one or more partners in the program; here, a *partner* of a program construct refers to another construct in the program whose sequences always show high similarity with those of this construct. For instance, the two loops in Figure 3 are partners of each other. These sequences form a subclass $C3.1$. They can be predicted from one to another, despite their irregularities. Some other sequences have several (e.g. less than 8) dominant values, as exemplified by Figure 5 (f). Although the time order in which these values occur appears random, knowing these dominant values may be still helpful in certain scenarios. They form the class $C3.2$. The other sequences in $C3$ constitute $C3.3$, which are considered as unmanageable in this work. Figure 5 (g) shows such an example.

The sequences collected in our experiment can be well mapped into the taxonomy. The seven rightmost columns in Table 1 show the distribution of the loop sequences among the classes. Such categorization lays the foundation for studying inter-sequence correlations, as the following sections will show.

Cross-Run Stableness of Patterns The last but not least property of the behavior sequences we observe is that while the sequences of a program construct may vary much across different runs in value and length, but not in shape or pattern. Figure 6 shows the five sequences of three loops in three programs. The sequences of different loops show large differences in shape. However, they all clearly exhibit certain shape stableness in the sense that the sequences in different runs appear to have similar shapes and patterns, even though the differences in the program inputs cause considerable variations in the sequence length and height of the curves. These loops are not exceptions. We say that a loop is *a stable loop* if all its sequences fall into a single category other than the unmanageable category C3.3 in the taxonomy. Of all the 11 programs, seven have more than 88% of their loops being stable. *Sphinx3* has 83% stable loops, and the rest all have more than 50% stable loops (64% for *sjeng*, 56% for *mcf*, and 50% for *parser*.) A detailed analysis shows that the main reason for the relatively low percentage of the three programs is not unstableness across runs but the irregularity of the sequences. Recall all loops in C3.3 are irregular. They are all automatically labeled as non-stable loops, even though stableness can often be seen in these loops: All their sequences tend to be irregular in all runs.

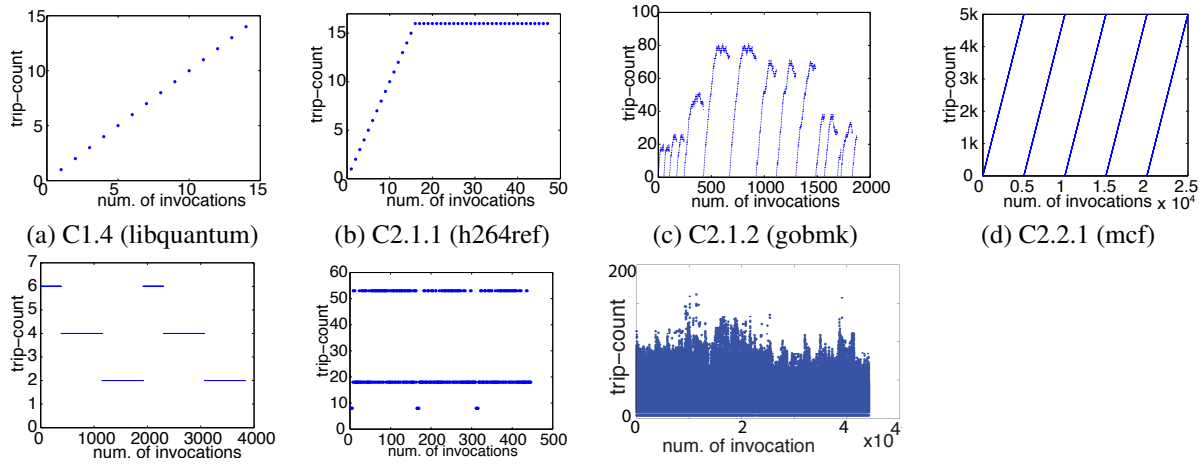


Figure 5. Sequence categories (given in Figure 4) and some examples that appear in the collected loop trip-count sequences of the programs (names in brackets) listed in Table 1.

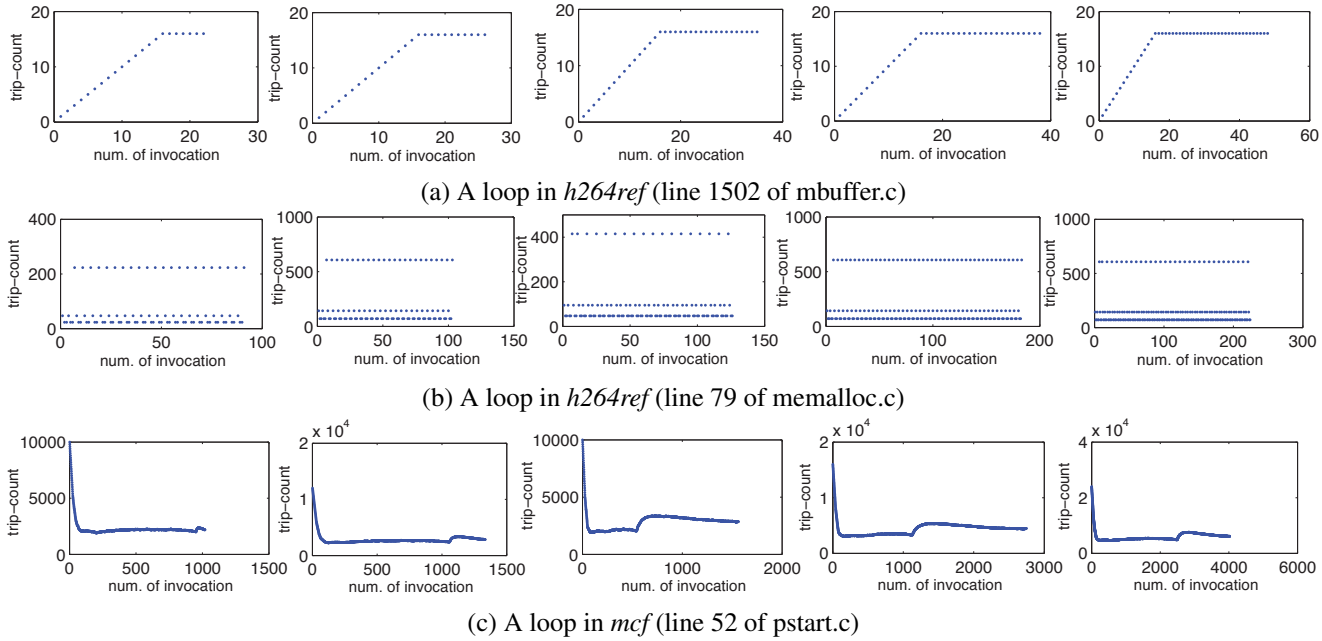


Figure 6. Trip-count sequences in five different runs.

The cross-run stableness reinforces the divide-and-conquer idea for addressing sequence complexity, in the sense that if we treat program constructs category by category, we need not worry about the cases that a construct may belong to different categories in different runs. Moreover, the cross-run stableness suggests the promise of using offline training for construct classification and correlation analysis.

2.3 Measuring Inter-Sequence Correlations

After achieving some understanding of behavior sequences, we are ready to examine the correlations among the sequences. One option for measuring the sequence correlations between two constructs is to compute the standard correlation coefficients between the behavior sequences of the two constructs. It cannot meet our needs because these coefficients capture only certain kinds of relationships (e.g., linear relationship by Pearson's coefficient or monotonic relationship by Spearman's coefficient) between the two construct's behaviors. Recall that the main purpose of studying the correlation is to enhance behavior prediction, useful relationships for which are apparently not only linear ones.

Considering the ultimate usage, we choose to use predictive capability to measure the strength of correlations. Construct A strongly correlates with construct B if A's behavior

sequences, although showing certain variations, can be accurately predicted from B's behavior sequences. If the prediction accuracy of construct A's sequences is higher from construct B than from construct C, we say A has a stronger correlation with B than with C.

This metric suggests that a predictor needs to be built between construct sequences to get the prediction accuracy, and the better the predictor is, the closer the obtained accuracy reflects the strength of the correlation. Creating the best predictor is difficult if ever possible. But an important insight is that because the best predictor gets no worse accuracy than a non-perfect predictor, a positive conclusion on the existence of a correlation by a non-perfect predictor must be correct. Driven by this insight, we develop the following correlation-based predictor.

2.4 Sequence Predictor

We name the predictor a *sequence predictor* because its output is a behavior sequence rather than the value of the next instance as most existing predictors output. The much larger scope of prediction is a result of its exploitation of inter-sequence correlations. Although the predictor is designed for examining the existence of inter-sequence correlations, it has a much broader usage as Section 3 will show.

The design of the predictor is based on the taxonomy and cross-run pattern stableness presented in Section 2.2. As Figure 7 shows, the predictor is built by offline training and is used for online prediction. The offline training is a synergy of intra-sequence pattern recognition and inter-sequence correlation analysis. It happens on the sequences collected on many training runs. It first recognizes the sequence pattern of a loop, and represents its sequences with a few pattern parameters (Section 2.4.1.) Working on the concise representation, it then uses statistical analysis to find the constructs related with one another, and builds predictive models among them (Section 2.4.2.) The models lead to large-scope sequence predictions during run time (Section 2.4.3.)

2.4.1 Pattern Recognition

The goal of this step is to categorize a construct into one of the classes in the pattern taxonomy, and then represent each sequence with a feature tuple. The tuple contains the class number of the construct and its corresponding pattern parameters. We start by explaining the concept of feature tuples.

Feature Tuples A feature tuple offers a way to concisely represent a sequence. It contains all important characteristics of a sequence such that from it, the sequence can be regenerated easily (with few exceptions.) Every non-C3.3 sequence can be represented by a feature tuple: $(c, \vec{p})$, where, c is the number of the leaf class in the taxonomy tree that the sequence belongs to, and $\vec{p}$ is the *pattern vector*, consisting of the parameters of the pattern of that class. Table 2 summarizes the format of the pattern vectors of each major class, where, $f(n)$ is one of the primary patterns listed in category C1, $\{0, logn, n^{0.5}, n, n^2\}$, and n is the instance number. For example, the pattern vectors for the sequence in Figure 5 (a) is (17, 0, 1), where, according to the second column in Table 2, 17 corresponds to the length of the sequence, and the other two elements come from the pattern of the sequence, *trip count = number of invocation*. Apparently, with such a vector, we can easily regenerate the entire sequence. In the same vein, the pattern vector for the sequence in Figure 5 (b) is computed as (9, 1.4, 32, -1, 20, 1.1, 22, 0), where the first four elements correspond to the first phase in the sequence with the second number, 1.4, indicating that the phase is in subclass C1.4 having a linear pattern; the other elements correspond to the second phase, which has a constant pattern.

Recognition Algorithm Figure 8 presents the algorithm for determining feature tuple, $(c, \vec{p})$. Each iteration of the outer loop of the function *seqClassify* processes one construct. It first calls *seqClassify_* to classify each sequence of the construct, and then checks whether all sequences of a loop belong to a single class other than C3 (by invoking the function *hasDiffClass()*). If so, it has finished processing that construct. Otherwise, it checks whether the construct belongs to C3.1 by trying to find a partner of it through comparisons between its sequences and the sequences of other loops whose IDs are smaller than this loop's. (Loop IDs are given in order of their first invocations in a typical run; observations on a loop with a lower ID may be hence used to help predict the behavior of a loop with a higher ID as shown in Section 2.4.3.) If it finds a partner for a construct currently labeled as C3.2, it relabels it to C3.1; we give C3.1 precedence because the partnership usually gives better prediction accuracy according to our experience. Other constructs are labeled to C3.3.

We briefly explain how a sequence is classified by the function *seqClassify_*. The function first applies regression to the sequence by invoking the *curveFit* function, which uses the standard curve-fitting method [13] to try to fit the sequence with each of the five primary patterns (i.e., the five patterns in class C1.) If the fitting error of the best of the five is smaller than a threshold (e.g., 10%), the sequence is considered as a sequence in the corresponding sub-class in C1, and the fitting result is taken as its pattern vector. Otherwise, *seqClassify_* tries to see whether the sequence has repetitive phases by invoking the function *repPhaseTest*. At a positive answer, it further checks the patterns of the first phase to determine which sub-class of C2.2 suites the sequence. At a negative answer, it invokes the function *phaseDet* to determine whether the sequence consists of non-repetitive phases, and identifies the patterns in each phase if so. Otherwise, it checks whether the sequence contains several dominant values through the function *domValueDet*, which simply examines whether the n most frequent values in the sequence together cover over 99% of the sequence. If so, it labels the

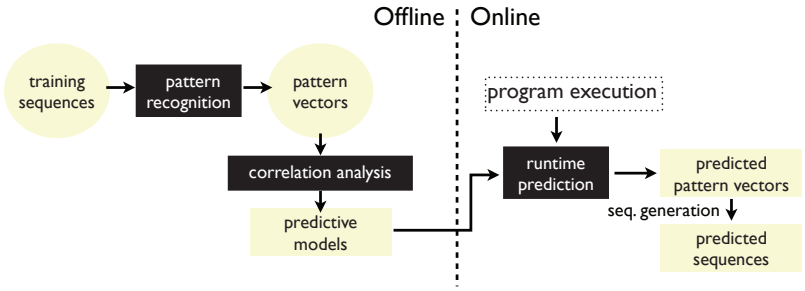


Figure 7. The three-step approach to enabling sequence prediction.

Table 2. Formats of pattern vectors

class	C1.1–C1.5	C2.1.1	C2.1.2	C2.2.1	C2.2.2	C2.2.3	C3.1	C3.2	C3.3
format	(l, p_1, p_2)	$(l_1, c_1, p_{11}, p_{12}, l_2, c_2, p_{21}, p_{22}, \cdots)$	-	(l, n, c_1, p_1, p_2)	$(l, n, l_1, c_1, p_{11}, p_{12}, l_2, c_2, p_{21}, p_{22}, \cdots)$	(l, n)	(p)	$(l, v_1, v_2, \cdots)$	-
note	l: seq. length; $p_1 + p_2 * f(n)$.	l_i: length of phase i; c_i: class of phase i; $p_{i1} + p_{i2} * f(n)$ for phase i	-	l: phase length; n: # of phases; c_1: class of phase 1; $p_1 + p_2 * f(n)$ for phase 1.	l: phase length; n: # of phases; c_i: class of phase i; $p_{i1} + p_{i2} * f(n)$ for phase i.	l: phase length; n: # of phases.	p: partner ID.	l: seq. length; v_i: the ith most frequent value.	-

Note: $f(n) \in \{0, logn, n^{0.5}, n, n^2\}$

sequence with C3.2. The proper value of n depends on the use of the prediction. Our experiment sets it to seven.

The rightmost column of Figure 8 outlines the two phase-detection algorithms. Unlike many existing algorithms [23, 24], the two algorithms distinguish the detections of repetitive and non-repetitive phases. It is initially for the need of sequence prediction, but later we find such distinction also simplifies the detection problem.

The function *repPhaseTest* uses a simple but effective approach we design to detect repetitive phases. Its basic idea is that for a sequence consisting of K repetitions of a phase, we should get k mutually similar segments if we cut the sequence into k even-sized partitions as long as k is an integer divider of K. The difficulty is to find a good k value. An insight taken by the function is that checking prime numbers suffices because any integer is the product of a series of prime numbers. If all prime numbers below a number (say m) are checked, it is guaranteed to cover all possible values of K whose smallest prime divider is no greater than m.

The function *phaseDet* is a simplified version of the wavelet-based phase detection algorithm proposed by Shen and others [23]. The second-level differentiation is equivalent to a second-level Haar wavelets transformation. It helps expose the drastic form changes in the sequence, which typically suggest phase boundaries (as illustrated by Figure 5 (e)).

2.4.2 Model Construction

After the sequences are represented with pattern vectors, the second step for building sequence predictors is to construct predictive models among the pattern vectors of different constructs. It uses a regression framework, working on the pattern vectors of all the sequences in the training set. Figure 9 outlines the high-level algorithm. At the core is a function "corRegress", which conducts regression between X and y, and reports the best model and regression error. The regression models it tries include the standard LMS for linear relations and Regression Trees for non-linear relations—two efficient models with broad applicability and interpretable results [13].

```
// inputs:
//   S: all sequences
//     S.nseq[i]: # of seq. of behavior i
//     S.seq[i]: set of seq. of behavior i
//   B: # of behaviors
// output: rst
//   rst[i][j].class: class of the jth seq.
//                    of behavior i
//   rst[i][j].pv: pattern vector of the
//                 jth seq. of behavior i

function seqClassify(S, B, &rst)
 for (i=0; i<B; i++)
  // classify each seq
  for (j=0; j<S.nseq[i]; j++)
   seqClassify_(S.seq[i][j], rst[i][j]);
  end for

  if (hasDiffClass(rst[i]) ||
      rst[i][0].class=="C3.2" ||
      rst[i][0].class==null)
   class3.add (i);
  end if
 end for

 // deal with class 3 behaviors
 for (i=0; i<class3.size; i++)
  b = class3.beh[i];
  for (j=0; j<i; j++)
   if (isPartner(S.seq[i], S.seq[j]))
    rst[b].class = "C3.1";
    rst[b].pv = j;
   end if
  end for
  if (rst[b].class==null)
   rst[b].class = "C3.3";
  end if
 end for
end
```

```
// classify s to C1, C2.2, C2.1, C3.2, or others
function seqClassify_(s, &rst)
 // C1
 rst1 = curveFit(s);
 if (rst1.minErr < H1)
  rst.class = "C1."+rst1.bestModel;
  rst.pv = getPatternVec(rst1);
  return;
 end if

 // C2.2
 rst2 = repPhaseTest(s);
 if (rst2.yes)
  rst3 = checkPhase1(s, rst2);
  rst.class = "C2.2."+rst3.class;
  rst.pv = getPatternVec(rst2, rst3);
  return;
 end if

 // C2.1
 rst4 = phaseDet(s);
 if (rst4.yes)
  rst5 = modelEachPhase(s,rst4);
  rst.class = rst5.pure? "C2.1.1":"C2.1.2";
  rst.pv = getPatternVec(rst4, rst5);
  return;
 end if

 // C3.2
 rst5 = domValueDet(s);
 if (rst5.yes)
  rst.class = "C3.2";
  rst.pv = rst5.values;
 end if
end
```

```
// test if s consists of repetitive phases
function repPhaseTest(s)
 rst.yes=0; rst.phN = 1;
 for (i=0; i<PR.size; i++)
  dif[i] = avg disparity among PR[i]
           even-size partitions of s;
  rdif[i] = dif[i] normalized by mean
            value of partitions
 end for
 m = findMin(dif);
 if (dif[m]<H2 || rdif[m]<H3)
  rst.yes = 1;
  k = s.size/m;
  // recursively minimize  phase granularity
  rst1 = repPhaseTest(s[0:k-1]);
  rst.phN = m*rst1.phN;
 end if
 return rst;
end


// test if s consists of clear phases
function phaseDet(s)
 phaseN = 0;
 dif2 = second level differentiation of s
 m = mean(dif2);
 d = std(dif2);
 for (i=0; i<dif2.size; i++)
  if (dif2[i] > m+d)
   phaseBounds[phaseN] = i;
   phaseN++;
  end if
 end for
end
```

Figure 8. Algorithm for sequence pattern recognition.

```
// A: the training data set
for each construct b
 for each construct b' that b'.id<b.id
  for each dimension d of b's pattern vector
   Let y be a vector containing all values of d in A
   Let X be a matrix containing all pattern vectors of b' in A
   Do regression:  corRegress(y, X, err, model);
   if (err < minErr)
    minErr=err; b.partner[d]=b'; b.model[d]=model;
   end if
  end for
 end for
end for
```

Figure 9. Sequence correlation detection.

The regression error is calculated through *10-fold cross validation* [17]. It uses 90% of X and y for model construction and the remaining 10% for testing. It repeats this process 10 times, with a different partition between the two sets each time. The average error is taken as the regression error for that model.

As shown in Figure 9, after this step, each construct gets a list of predictive models, which each corresponds to one element in its pattern vector. In addition, the construct gets a list of partners. The ith partner is a construct whose pattern vector strongly correlates with the ith dimension of the pattern vector of the current construct.

2.4.3 Runtime Prediction

Using the constructed predictors for runtime prediction is straightforward for most classes of sequences. For instance, suppose the algorithm has recognized that a loop L1 is the partner of another loop L2 in terms of their sequence correlation. Let $f()$ represent the predictive model mapping from the pattern vectors of L1 to those of L2. In a new run, after a number of invocations of L1, its sequence pattern vector is computed from the observed iterations of those invocations. Putting the vector into the predictive model $f()$ then produces the pattern vector of L2. From that vector, the whole sequence of L2's behavior can be immediately generated. The prediction exhibits a large scope and does not need to wait for the target construct to occur in the current run, which are appealing properties over locality-based predictors.

Some special treatment is needed during the sequence prediction for classes C2.1.2, C2.2.3, C3.2, and C3.3 due to their complexities. For sequences in class C2.2.3, the prediction starts after the first phase finishes. Based on the repetitiveness of the phase, the prediction is simply $n-1$ copies of the first phase, where n is the number of phases, predicted using the offline constructed predictive models. For these sequences, the proactivity is not complete, but usually high still.

For sequences in class C3.2, as the order in which the dominant values appear is hard to determine, the dominant values are predicted, but no sequences are generated. For sequences in class C2.1.2 or C3.3, no prediction is conducted. From Table 1, we see that for most programs, only a small portion of loops fall into those special classes.

The runtime prediction of pattern parameters is typically lightweight, involving just the computation of a linear expression or a traverse of a several level regression tree. Compared to many runtime optimizations (e.g., JIT compilation), the overhead is negligible.

2.5 Prediction Accuracy: Confirming Existence of Correlations

We apply the sequence predictor to the loop trip-count sequences of the programs mentioned in Section 2.1. A reasonable accuracy will help confirm the existence of inter-sequence correlations.

Before detailing the result, we explain the calculation of the accuracy of a predicted sequence. One option is to compare, from left to right, each pair of corresponding instances in the predicted and true sequences. But if one instance is missing in the predicted sequence, the misalignment may make the following instances appear erroneous even if they are correctly predicted. In practice, some difference in sequence length may not impair the use in optimization much. We choose to use real sequence length during the generation of a predicted sequence, and present the accuracy of sequence length prediction separately. Ten-fold cross validation is used so that testing runs differ from training runs.

Figure 10 (a) reports the average accuracies. The accuracy of a predicted sequence is defined as $\frac{1}{n}\sum_{i=1}^{n}(1 - |v'(i) - v(i)|/max(v'(i), v(i)))$, where n is the length of the true sequence, and $v(i)$ and $v'(i)$ represent the values of the i'th instance in the predicted and true sequences respectively. The dominator ensures a 0-100% range. The first two bars of each benchmark show the accuracy of the predicted sequence length and pattern parameters. Most programs' are above 90% on both, including the largest software, *gcc*. The program *sjeng* gives the lowest accuracy because of its large number of branches and recursive function calls.

The third bar of each benchmark shows the accuracy of predicted sequences. Most programs' are above 85%; the overall average is 91%. To put the result into context, we apply a pure locality-based predictor to predict the sequences. It generates the sequence of a loop purely based on its earlier invocations. The design maintains a prediction scope comparable with the sequence prediction for a fair comparison. The accuracy is shown by the rightmost bar of each benchmark. On average, the accuracy is 76%, 15% lower than the sequence predictor's result.

Because we have 20 inputs for every benchmark and the prediction accuracies on their corresponding runs are often different from one another, it is worthwhile to examine the distributions of the prediction accuracies. We show the distributions through boxplots in Figure 10 (b). In each boxplot, the short line segments at its top and bottom show the accuracy range, and the middle square covers the medium 50% of the accuracies. A large gap shows up between the two boxplots of every benchmark. Statistical significance test (T-test) gives all zero p-values, further confirming the consistent, significantly better accuracies from sequence prediction than from locality-based prediction, indicating the value of the correlations exploited by the sequence predictor.

The prediction accuracy by the sequence predictor reflects the broad existence of inter-sequence correlations. But two important questions remain open. Are the correlations strong enough for actual uses in program optimizations? Is the sequence predictor effective enough for translating the correlations into what the optimizations need? We implement two uses to answer these questions, as described next.

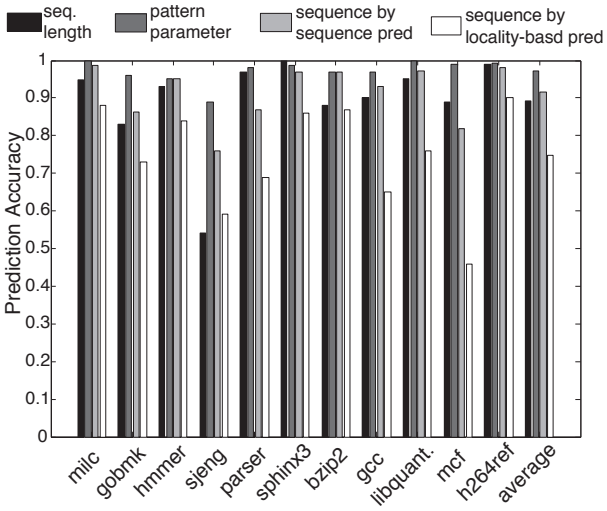
3. Uses of Correlation-Based Sequence Predictors

In this section, we first use dynamic function version selection to demonstrate the usefulness of the correlation-based prediction for optimizing some large applications. We then show its uses for loop importance estimation on SPEC CPU2006 benchmarks written in C. Section 4 will briefly discuss uses in other contexts.

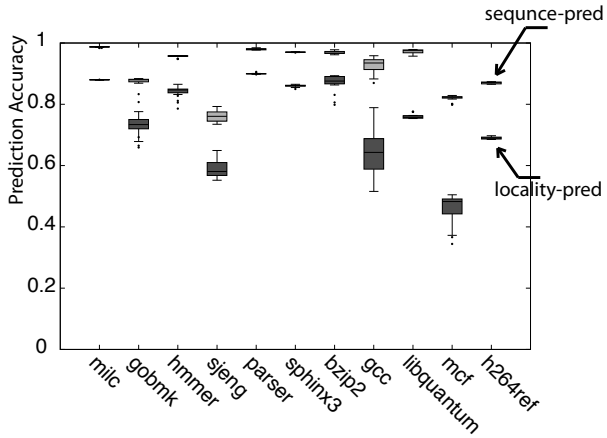
3.1 Dynamic Function Version Selection

Dynamic function version selection is a kind of runtime optimization. For a given function, multiple versions are generated during offline compilation. They perform well in different scenarios. Dynamic version selection predicts the best version for an upcoming invocation of the function. This optimization is especially useful for utility programs whose input consists of a sequence of different tasks, such as a compiler compiling many different functions, a parser parsing many different sentences. These tasks stimulate different behaviors of the program, hence creating the needs for dynamic version selection.

In our experiments, we use the feedback-driven compilation of GCC to create three versions for the functions in each of the benchmarks. The feedback-driven compilation uses the profile of a training run of a program to reoptimize the program. The three versions are the results of the feedback-driven compilation when three different training runs on different inputs are used.



(a) Average accuracy of the predicted sequence lengths and pattern parameters by the sequence predictor (left two bars), and average accuracy of predicted sequences (right two bars.)



(b) Boxplots showing the distributions of the sequence accuracies. Upper box: by sequence predictor; lower box: by locality-based predictor.

Figure 10. Average accuracy (a) and statistical distributions of the accuracies of the predicted sequences (b).

We use our correlation-based prediction to help dynamic version selection. The idea is to take its predicted loop sequences as indications of the behaviors of the upcoming invocation of a function. For example, there is a call to function *foo* in a utility program and this function contains five loops. To apply dynamic version selection to that call, a call to a version selection procedure is inserted before it. Using the predictive models built offline, the procedure predicts what trip-count sequences the five loops of *foo* will produce in the coming invocation. It then derives a 5-element vector with each element corresponding to one loop, equaling the sum of its predicted trip-count sequence. We call the vector the predicted *behavior signature* of the coming call. There is a behavior signature of *foo* prepared for each version during the creation of the three versions. By comparing the predicted behavior signature with them, the dynamic version selection procedure selects the version whose signature is closest to the predicted one. The closeness is defined by $\sum_{i=1}^{5} |v'_i - v_i|/(1 + v'_i)$, where v'_i and v_i are the elements of the predicted and reference signatures. The normalization by $(1 + v'_i)$ reduces biases by the absolute trip-count values (adding one avoids division by 0.)

We compare with two alternatives. The first is static compilation at the highest optimization level. The second is an implementation of a prior dynamic version selection scheme [8], which uses locality-based prediction, working as follows. During the initial several invocations of a function, the technique tries a different version at each invocation. By comparing the running times of the several invocations, it selects the version with the shortest running time and uses that version for a number of upcoming invocations of the function. For better adaptation, the sampling of different versions happen periodically. We experiment with four sampling period lengths: 6, 12, 24, and all invocations.

We next report our findings on six utility applications. They mostly come from real-world applications with thousands or hundreds of thousands of lines of source code. They are representatives of the domains ranging from natural language processing, program compilation, scripting language interpretation, to data mining. One of them, *DataMine*, is a CPU-GPU mixed program and has some special aspects for version selection. We postpone it to the end of our discussion. In the experiments, we focus on only one or several major functions that consume a significant part of the total running time of the program. The reported performance is the average of five runs on the Xeon E5310 machine with GCC 4.4.1 installed. Figure 11 summarizes the result.

Sleator-Temperley Link Parser v2.1 (Parser) According to SPEC2K web site, this program is a syntactic parser of English, based on link grammar with about 60000 word forms. It has 11,391 lines of C code. We use its batch-processing mode, in which, an input contains a number of sentences for parsing. Different sentences trigger different behaviors of the functions in the program. The three training files we use each contain 2, 5, 20 sentences respectively; although the sentences in one file are of similar length and complexity, the sentences in different files differ significantly. The testing input we use consists of 10,000 sentences of various length and complexity.

The parsing of one sentence goes through a few stages. The correlations we find are between loops in the first stage and those in the other stages. The dynamic version selection is on these later stages. The correlation-based prediction yields 89% average prediction accuracy on the signatures. Because finding the version with the closest signature is a qualitative problem, version selection has a quite large error tolerance. The prediction gives a perfect version selection

accuracy, producing a speedup of as much as 17.7% compared to the best static compilation result. In comparison, the locality-based dynamic version selections fail to find the best versions at more than half of the invocations, producing 10.9% speedup.

GNU Compiler Collection v3.2 (GCC) As a widely adopted compiler, GCC, written in 517K lines of code, takes C or C++ source code as its input and compiles the contained functions one by one. The compilation of different functions may trigger the invocations of different procedures in GCC and stimulate different behaviors of GCC. The three training inputs we use consist of loop-intensive, branch-intensive, and straight-line code respectively. The test input contains 40,299 lines of C code with 322 functions of various kinds. The compilation of a function goes through a number of compilation passes. The correlations we find are between the loops in the earlier passes and later passes. The accuracy for function signatures is 90% and the best version can be found for 93% of all invocations. The resulting speedup is as much as 9.4% over the static compilation result, 4.4% more than the locality-based dynamic version selections.

SQL Database Engine SQLite v3.7.9 (Sqlite) Sqlite is one of the most widely deployed SQL database engines in the world [1]. It has 110K lines of C code. We use three types of "select" queries as training inputs, with some containing a simple single selection condition, some containing multiple conditions with "AND" and "OR" relations, some containing conditions involving numerical comparison and computations. The differences in the three types of queries trigger the invocation of different parts of Sqlite. As a result, the three versions produced by the feedback-driven compilation of GCC are quite different from one another, each suiting one type of the queries much better than the others and the statically optimized version. Compared to the statically optimized version, the three versions show 34–65% speedup on the corresponding types of queries.

The query file for testing consists of 10,000 queries with all three types of queries included. The sequence prediction-based dynamic selection is able to choose the best versions for all the queries, producing an average speedup of as much as 52%, 12% more than the locality-based version selection.

Bzip2 v1.0.3 and Sphinx-3 Bzip2 is a widely used data compressor in 8,293 lines of code. Its inputs may be files of different kinds. We use an image, a novel, and a set of XML files as training inputs to create three versions. We use a file in rich format containing all three kinds of content as the testing input. Our version selection focuses on a buffer compression function. The locality-based selection give similar performance as our approach, both less than 1.8% away from the static optimization result. We find that even the perfect version selection gives only 2% speedup. An analysis of the source code shows that the program separates the input into equal-sized buffers. The compression on the buffers behave quite similarly, regardless the type of input, hence the insignificant speedups. Sphinx3 is a widely known speech recognition system with 23K lines of code. It shares similar properties as Bzip2, showing insignificant speedup in all cases.

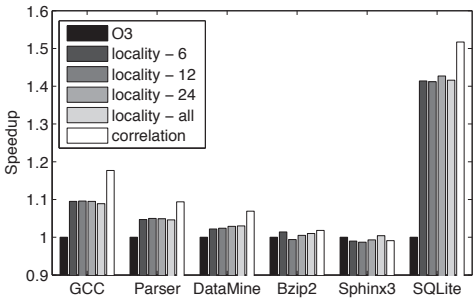


Figure 11. The performance brought by locality-based and correlation-based version selection. Baseline is static optimization result.

Customer Relationship Management (DataMine) DataMine is a program we derived for customer relationship management. Figure 2 (a) has outlined its code skeleton. It uses K-means to cluster customers into a number of groups. It then loads the customer activities if necessary and processes the customer information of each group to extract some high-level knowledge about the customer group.

The version selection for this program differs from the other applications we have described. This program is intended to run on heterogeneous computing systems. It comes with two versions of the central workload-handling function *process()*, one for CPU, the other for Graphic Processing Units (GPU), determined by the boolean variable *bUseGPU*. The dynamic version selections try to select the appropriate version for each invocation of the function. The selection mainly depends on the amount of computation involved in the workload handling, determined by the size of the cluster but also the fraction of the customers in the cluster that have subscribed the web service for more than 12 months. The relation between the amount of computation and the best version to use has been determined through offline profiling.

The detected correlations are between the loops in the clustering (including a loop in the *loadActivity* function) and the loops in the core function *process* (CPU version). The predicted signature of that function reflects the amount of computation in the next invocation of the core function. The test has 1000 clusters with size ranging from 15 to 354,286. The accuracy of signature prediction is 100%, leading to perfect version selection. The speedup is 8.5% over the all-CPU execution and 7.6% over the all-GPU execution, significantly outperforming all locality-based versions. (In Figure 11, the all-GPU is used as the baseline.)

Runtime Overhead The overhead of correlation-based version selection consists of three parts: recording the trip-count sequence pattern parameters of some predictor loops,

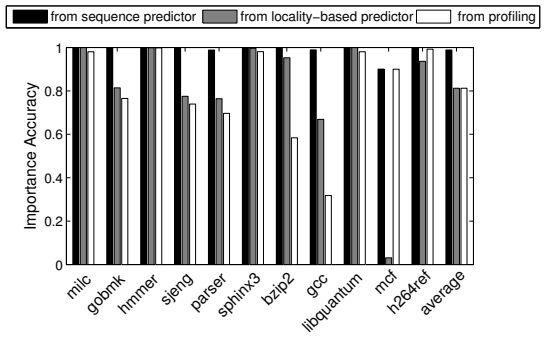


Figure 12. Prediction of the importance of loops.

applying the offline built models (a linear function or a several-level decision tree) to produce the predicted trip-count sequence of the target loops, and reducing the sequences to signatures. The second and third parts involve only a few, simple calculations and have negligible overhead compared to the large number of operations in the optimization target. The amount of overhead of the first part depends on the types of the predictor loops. If their sequences are composed of primary patterns, only a few samples (i.e., trip-counts) are sufficient for computing their pattern parameters (e.g., two samples are enough for a linear pattern.) For Class C3.1, however, the whole loop sequence needs to be recorded. An important insight for minimizing the overhead is that for predicting the correct version to use, it is usually unnecessary to predict the trip-counts of all loops. We can construct the signature by selecting only the loops that do not need detailed instrumentation for prediction and meanwhile have large variances in their sequence values across different runs.

3.2 Prediction of Loop Importance

In this experiment, we use the predicted behavior sequences to estimate the importance of a loop in a run, which is defined as its total trip count divided by the sum of all loops' trip counts. Such information is essential for guiding loop optimizations to concentrate on important loops. Meanwhile, as it reflects the relative frequency of different memory references and functions, it is beneficial for guiding prefetching and function inlining as well [2].

We compare three methods on all the manageable loops (i.e., non-C3.3 loops.) The first computes the importance directly from our predicted loop trip-count sequences. The second is through a direct extension of locality-based prediction. It computes the importance based on the first invocations of all loops in the current execution. This extension maintains a prediction scope comparable with the other two methods for fair comparison. The third is a pure offline profiling-based approach, using the profiles in the *test* run for estimation of real runs.

Our experiment uses all the C programs in SPEC CPU2006 that Section 2.1 has described. Figure 12 reports the estimation accuracy on the *ref* runs of the benchmarks. The accuracy is defined as $(1 - \sum_i(|m[i] - m'[i]|)/2)$, where, $m[i]$ and $m'[i]$ are the real and predicted importance of the ith loop. Division by two is to normalize the accuracy to the range of 0–100%. The offline profiling approach is subject to input sensitivity, hence showing the lowest accuracy. The pure locality-based method achieves high accuracy on six of the eleven benchmarks. On average, the sequence prediction gives 98% accuracy, 17% higher than the locality-based method and the profiling method.

4. Discussion

In this paper, we have used loop trip-count sequences to demonstrate the existence of inter-sequence correlations in program dynamic behaviors, and showed the feasibility of sequence prediction and its potential for program optimizations. Loops are among the most important constructs in programs. But there are many other kinds of program constructs and behavior sequences, such as function return values, referenced memory addresses, data reuse distances, and so on. The inter-sequence correlation study described in this paper and the ensuing sequence predictor construction are potentially extensible to these types of behaviors. They may open up many new opportunities, not just for program-level optimizations, but also for optimizations at other levels of software execution stacks. At the Operating System level, for instance, the enhanced scope, timing, and accuracy of program behavior prediction may better reveal the resource requirement of an application in its different phases, and hence better guide resource provision in data centers [12] and job (co-)scheduling on non-uniform or heterogeneous architecture [28, 30]. At the architecture level, the prediction may better help dynamic voltage scaling and cache partition/reconfiguration to avoid local optimum traps with its large prediction scope. We expect that the promising results observed in this current work will help stimulate a broader interest of the community in exploring and exploiting inter-sequence correlations and sequence predictors in these other settings.

Directly characterizing the data flowing into a function may also help predict function behaviors, but automatically doing so is difficult, subject to the complexity of data and needs for domain knowledge. Correlation-based prediction is a complement rather than replacement of locality-based prediction. The latter is useful for refining prediction results with local information. How to best combine them is worth futher exploration.

5. Related Work

Program behavior prediction has been studied broadly. Examples include branch prediction [27], load value prediction [4], and return value prediction [15, 21]. These predictors are typically based on the history instances of a behavior. Some consider only recent instances, such as the last

(N) value method [3, 20], stride method [10, 22]. Some concern calling contexts, such as finite context method [11, 22]. Some further consider the arguments of functions, such as parameter stride [15], memorization [21]. Some others combine multiple types of predictors into one [5, 21]. They are all based on locality of behaviors. There are some studies on behavior histogram prediction [9, 29]. Although more complex than an instance, a histogram differs from sequences in that it has no temporal order.

Program phase prediction [23, 24] also tries to reveal some large-scope behavior patterns. However, its goal is to predict phase shift, rather than a behavior sequence. Phase shift is one of the many patterns exploited in sequence prediction. Some recent studies try to enable proactive behavior prediction by drawing the heuristics from program inputs [25]. However, their prediction target is still one instance of a behavior (e.g., the appropriate optimization level of a Java method).

Recent years have seen a body of work that uses Machine Learning (ML) techniques to assist program optimizations. Examples include prediction of the optimization levels for a Java method [6], prediction of suitable parallelization schemes [26], and so on. This current work is complementary to these previous techniques. These studies typically build predictive models mapping from program code features (e.g., portions of various instructions contained) to optimization decisions, rather than exploit inter-sequence correlations. We are not aware of a prior work in those studies that predicts large-scope behavior sequences. Such predictions are important for optimizations that rely on large-scope, dynamic behaviors of a program. For instance, for the dynamic version selection in Section 3.1, the suitable versions to use at an invocation of a function depends on how it will run in that invocation rather than what code it contains. The previous ML-based method [6] is not applicable for such uses. On the other hand, the output from our sequence predictors may be used to enrich the feature vectors used in constructions of ML-based optimization models, as exemplified by our dynamic program version selection experiment.

The correlation analysis of this current work is enlightened by a previous study [16], which uses correlations to predict average values of a behavior, rather than a sequence. Their study is oblivious to the sequence complexities covered in this work. Some branch predictors use correlations among branches. But the exploitation has been both implicit and simple, in a manner that hardware can accommodate. And the prediction target is usually the next branch rather than a large-scoped sequence of dynamic behaviors. Some prior studies have used compression tools, such as SEQUITUR, for finding hot path or data streams [7, 18]. Hot streams capture some often seen short sequences of discrete events (e.g., variable c is likely to follow an access sequence $b\ d\ e$), rather than large-scoped sequences of numerical behaviors (e.g., the loop trip-count sequences in this paper.) Overall, we are not aware of a prior study that analyzes the taxonomy of program behavior sequences, or systematically exploits inter-construct correlations for large-scoped prediction of a sequence of numerical program behaviors.

6. Conclusions

This paper presents an exploration on inter-construct correlations. It examines their existence, and creates the first hierarchical taxonomy for categorizing behavior sequences. It develops a new form of predictors, namely sequence predictor, to translate the correlations into large-scoped, proactive prediction of program behaviors. Experiments on loop trip-count sequences demonstrate the strength of inter-construct correlations, and the effectiveness of sequence predictors in producing accurate prediction of behavior sequences. This study concludes the broad existence of inter-construct correlations, reveals their large values in enhancing program behavior predictors' proactivity and scope, and demonstrates the promise of sequence predictors in opening new opportunities for program and system optimizations.

Acknowledgement

We owe the anonymous reviewers our gratitude for their helpful suggestions on the paper. This material is based upon work supported by the National Science Foundation under Grant No. 0811791 and CAREER Award, DOE Early Career Award, and IBM CAS Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation, DOE, or IBM.

References

[1] Sqlite. `http://http://www.sqlite.org/`.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006.

[3] M. Burtscher and B. G. Zorn. Exploring last n value prediction. In *PACT*, 1999.

[4] M. Burtscher and B. G. Zorn. Prediction outcome history-based confidence estimation for load value prediction. *Journal of Instruction-Level Parallelism*, 1999.

[5] M. Burtscher and B. G. Zorn. Hybrid load-value predictors. *IEEE Transactions on Computers*, 2002.

[6] J. Cavazos and M. O'Boyle. Method-specific dynamic compilation using logistic regression. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2006.

[7] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[8] P. Chuang, H. Chen, G. Hoflehner, D. Lavery, and W. Hsu. Dynamic profile driven code version selection. In *Proceedings of the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*, 2007.

[9] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *PLDI*, 2003.

[10] F. Gabbay. Speculative execution based on value prediction. Technical Report 1080, Israel Institute of Technology, 1996.

[11] B. Goeman, H. Vandierendonck, and K. de Bosschere. Differential fcm: Increasing value prediction accuracy by improving table usage efficiency. In *HPCA*, 2001.

[12] S. Govindan, J. Choi, B. Urgaonkar, A. Sivasubramaniam, and A. Baldini. Statistical profiling-based techniques for effective power provisioning in data centers. In *EuroSys*, 2009.

[13] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.

[14] J. Henning. Spec2000: measuring cpu performance in the new millennium. *IEEE Computer*, 2000.

[15] S. Hu, R. Bhargava, and L. K. John. The role of return value prediction in exploiting speculative method-level parallelism. *Journal of Instruction-Level Parallelism*, 2003.

[16] Y. Jiang, E. Zhang, K. Tian, F. Mao, M. Geathers, X. Shen, and Y. Gao. Exploiting statistical correlations for proactive prediction of program behaviors. In *CGO*, 2010.

[17] P. Jonathan, W. J. Krzanowski, and W. V. McCarthy. On the use of cross-validation to assess performance in multivariate prediction. *Statistics and Computing*, 10(3), 2000.

[18] J. R. Larus. Whole program paths. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.

[19] C. Lattner. *LLVM: An Infrastructure for Multi-Stage Optimization*. PhD thesis, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, 2002.

[20] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *MICRO*, 1996.

[21] C. Pickett, C. Verbrugge, and A. Kielstra. Adaptive software return value prediction. Technical Report 1, McGill University, 2009.

[22] Y. Sazeides and J. E. Smith. The predictability of data values. In *MICRO*, 1997.

[23] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *ASPLOS*, 2004.

[24] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *ISCA*, 2003.

[25] K. Tian, Y. Jiang, E. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *OOPSLA*, 2010.

[26] Z. Wang and M. O'Boyle. Mapping parallelism to multicores: a machine learning based approach. In *PPOPP'09: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 75–84, 2009.

[27] T. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *ISCA*, 1993.

[28] E. Z. Zhang, Y. Jiang, and X. Shen. Does cache sharing on modern cmp matter to the performance of contemporary multithreaded programs? In *PPoPP*, 2010.

[29] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems*, 31(6), 2009.

[30] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS*, 2010.