Space-Efficient Multi-Versioning for Input-Adaptive Feedback-Driven Program Optimizations

Mingzhou Zhou, Xipeng Shen[◇], Yaoqing Gao[*], Graham Yiu[*]

The College of William and Mary
[◇]North Carolina State University
[*]IBM Toronto Labs
mzhou@cs.wm.edu
[◇]xshen5@ncsu.edu
[*]{ygao,gyiu}@ca.ibm.com

Abstract

Function versioning is an approach to addressing input-sensitivity of program optimizations. A major side effect of it is notable code size increase, which has been hindering its broad applications to large code bases and space-stringent environments. In this paper, we initiate a systematic exploration into the problem, providing answers to some fundamental questions: Given a space constraint, to which function we should apply versioning? How many versions of a function should we include in the final executable? Is the optimal selection feasible to do in polynomial time? This study proves selecting the best set of versions under a space constraint is NP-complete and proposes a heuristic algorithm named CHoGS which yields near optimal results in quadratic time. We implement the algorithm and conduct experiments through the IBM XL compilers. We observe significant performance enhancement with only slight code size increase; the results from CHoGS show factors of higher space efficiency than those from traditional hotness-based methods.

Categories and Subject Descriptors D.3.4 [*Processors*]: code generation, optimization

General Terms Compiler, Performance

Keywords Program inputs, Function versioning, Feedback-driven program optimization

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-2585-1/14/10…$15.00.
http://dx.doi.org/10.1145/2660193.2660229

1. Introduction

Feedback-driven program optimization (FDO) is an important technique for enhancing program performance. By collecting the behaviors of some training runs of a program, FDO augments static code with some dynamic information of the program, helping compiler make better optimization decisions. It is a component available in most commercial C/C++/Fortran compilers (e.g., GCC, IBM XLC, Intel ICC), and has been widely used in industry software development [4, 19].

An important challenge to FDO is input sensitivity. A program's inputs can have a large variety. Often, different inputs may prompt the program to behave very differently. As a consequence, the code produced by FDO on one training run may work inferiorly on a different input [2, 10, 13, 20, 21, 23, 26]. Such input sensitivity is especially prominent in arising data-driven computing (e.g., business analytics), the data in which show increasing variety and complexity.

Multi-versioning is an approach that people have found effective for mitigating the problem [5, 10, 12, 25]. The idea is to create multiple versions of a program (or of some of its functions), with each version obtained by applying FDO on the profile collected on a run on a representative input. Some runtime version selection mechanisms are integrated into the code so that in production runs, on an arbitrary input, the appropriate version can be selected to use in that run.

An important limitation of multi-versioning is code size bloating. When the versioning happens at the whole-program level, the code size increases linearly with the number of versions. As modern applications and their input data both become increasingly complex and diversified, the base code size grows, and meanwhile, an even larger number of versions are required to fit the needs of various inputs. The issue is especially serious for portable and embedded devices for which, space is stringent and precious.

A natural solution is to be selective. Rather than creating versions at the whole-program level, one may create multiple versions only on some functions—an idea being adopted by most existing solutions. The designs of these solutions, however, have been ad hoc. They typically just select the functions that consume the largest portion of the program execution time to do mutli-versioning. The simple design suffices when the time distribution among functions is extremely skewed such that a small number of functions in a program consume most of the execution time. Our study finds that even though some functions are indeed hotter than others, such a hotness-based approach yields lots of space waste for some hot functions are not input sensitive while many input sensitive and important (even though not among the hottest) functions are not included in the versioning.

This paper presents the first systematic investigation into selective function multi-versioning, and uncovers multi-fold findings.

First, it formally defines the optimal selective function multi-versioning problem (OptMVP). According to the definition, the goal of OptMVP is to select, from a given set of versions of a program, a set of versions of each of its functions such that when they are put together, the resulting binary, among all possible choices that meet a given space budget, is able to provide the highest performance over all inputs (assuming the runtime uses these versions appropriately). The nature of the problem is a constrained optimization problem, with the space budget as the constraint, and the performance maximization as the objective. (Section 3)

Second, the paper provides a principled understanding in the complexity of OptMVP. It proves that the problem is NP-complete through a reduction from PARTITION, a classic NP-complete problem. To our best knowledge, this result is the first revelation of the complexity of the problem. The significance is that it will help the community avoid wasting time in finding algorithms to compute the optimal, and instead devote the efforts into the more promising direction of designing effective approximation algorithms. (Section 4)

Third, the paper gives a comprehensive discussion on the practical complexities in designing an approximation algorithm for OptMVP. One of the prominent complexities comes from the side effects of compiler optimizations. When FDO produces different executables on different profiles, it often applies different code transformations (e.g., inlining) and hence leads to different dynamic call graphs of those executables. The difference complicates the version selections. (Section 5.1)

Fourth, the paper presents Callgraph-based Holistic Greedy Selection (CHoGS), a simple yet effective approximation algorithm for OptMVP. The algorithms have several appealing properties. It assesses function versions in a holistic manner by examining the impact of a version to the entire program, rather than its own execution time. It is resilient to mismatches in dynamic call graphs and other complexities caused by FDO optimizations. Its quadratic time complexity makes it applicable to large programs (Section 5.2).

Finally, the paper reports the results of 11 programs from SPEC CPU 2000 and SPEC CPU 2006 with 10 inputs for each. It compares the performance of the executable produced by CHoGS and a hotness-based version selection, showing that CHoGS is much more effective in space-efficient versioning. On most of the benchmarks, CHoGS materializes over 87% of the full performance benefits of multi-versioning with only 10% extra space cost, while the hotness-based selection gives less than half of the benefits with that much space cost. To get the similar performance benefits, the hotness-based selection requires 4X to 8X more space cost than CHoGS needs.

Overall, the paper makes the following major contributions:

- **OptMVP.** To our best knowledge, this paper gives the first formal definition of the optimal selective function multi-versioning problem (OptMVP), and presents the first systematic study on it.
- **Complexity.** The paper, for the first time, reveals the computational complexity of the OptMVP, as well as the practical complexities in designing a solution for effective approximations.
- **CHoGS Algorithm.** The paper presents an effective approximation algorithm, CHoGS, which provides significantly higher space efficiency for multi-versioning than traditional methods.

2. Background and Scope

This section provides some necessary background on FDO and multi-versioning, along with the scope of this research.

FDO FDO consists of two stages. The first stage is called *profiling stage*, in which, monitoring the execution of the target program produces profiles that contain some observed runtime behaviors of the program. The second stage is called *reoptimization stage*, in which, the compiler reoptimizes the program based on the profiles. FDO may happen in both offline and Just-In-Time (JIT) compilations. In the former case, both stages happen before the final release of the program, and the profiling stage happens on some training runs of the program. Such a paradigm is common for traditional programming languages (e.g., C/C++). In the latter case, both stages happen during production runs. They are typically supported by a runtime system (e.g., Java Virtual Machine, Javascript engine) that is equipped with a JIT compiler.

In both cases, there is the space bloating problem when multi-versioning is used. In this work, we concentrate our investigation on the former case. JIT-based systems are more stringent on runtime overhead, but most of the complexities revealed in this work also exist on those systems.

Multi-versioning Multi-versioning is a way to increase the adaptivity of a program. In the context of FDO, multi-versioning consists of two components. The first is the creation of multiple versions of a code region, with each version specialized to a class of execution contexts. The second is a runtime selector, which selects the right version of the code region to use at execution time. The selector is often in form of some conditional statements that the compiler inserts into the program.

The benefits of multi-versioning depend on both components. The quality of the collection of versions for code regions determines the ultimate potential of the multi-versioning, while the quality of the runtime selector determines how much of the potential can be materialized. The focus of this study is on the first component. In experiments, it is necessary to isolate the concerns of the second component; we hence assume the presence of a desirable runtime version selector.

The code region to do multi-versioning can be at various granularity, ranging from a loop to a function to a whole program. The context for specialization can also be at various levels, ranging from a architecture to a class of inputs to a calling context.

In this work, we use function as the code region for versioning as it is readily supported by our compiler. Meanwhile, as input sensitivity is the main factor considered in this study, we focus on versioning at the level of program inputs rather than detailed calling contexts. So, in multi-versioning under our study, one version of a function is selected to use for all invocations of that function throughout an execution of the program on a particular input. Calling context-sensitive version selection is orthogonal to this study.

XLC Compiler This study is based on the XLC compiler, the commercial C/C++/Fortran compiler from IBM. The compiler has a built-in support for FDO. An FDO through XLC includes the following three steps:
Step 1: Compile the program with flag "pdf1". The compiler will generate a piece of binary code with some monitoring code inserted.
Step 2: Run the binary code on some input. This training run will produce a profile that records basic block frequencies, call edge invocations, and the values of some special variables.
Step 3: Recompile the program with flag "pdf2". The compiler will reoptimize the program based on the profile produced in Step 2, and generate a new piece of binary code of an often better quality.

The FDO in XLC consists of many sophisticated optimizations, such as function inlining, code layout transformation, loop optimizations, and so on. As the generated code is customized to the profile, it often works well on inputs similar to the one used in Step 2, but not necessarily on other inputs—the so-called "input sensitivity" problem. The default XLC has a preliminary support for addressing input sensitivity. It allows averaging the profiles from multiple runs on different inputs and then feeding the average profile to the recompilation in Step 3. The method often gives unsatisfactory performance, as Section 7 will show.

3. Problem Definition

This section gives a formal definition of the OptMVP.

Definition 3.1. Optimal Multi-Versioning Problem (OptMVP)

Given:

1. A program P with M functions. There are N different inputs to P, and N executables of P with the ith executable produced by an FDO compiler upon the profile collected on the ith input. E_i $(i = 1, 2, \cdots, N)$ represents the i^{th} executable. All executables are sound and complete, meaning that they keep the semantic of the original program, producing the same output on an arbitrary input as the original program does. Each function has one copy in each executable; some copies may be empty (e.g., when the function is inlined at every call site). $v_{i,j}$ represents the j^{th} version of the i^{th} function; $v_{i,j} \in E_j$. Let C represent the set of all versions of all functions in the N executables: $C = \{v_{i,j}; 1 \leq i \leq M, 1 \leq j \leq N\}$.

2. A given runtime version selector R which is able to select the best version of a function to use for a given input. Here, the best version is the version that gives the highest performance improvement with a unit of space cost.

Goal: find a subset of C, represented as $S \subseteq C$, such that when the members of S are assembled by R into an executable, X, the executable meets the following conditions:

1. It is sound and complete.

2. The space cost of the executable is within a given budget. That is, $\sum_{w \in S} l(w) \leq B$, where, $l(w)$ is the code size of a function in S, and B is the given space budget.

3. The executable gives the overall largest speedup—that is, the following is maximized among all possible choices of X:

$\sum_{i=1}^{N} T_{o,i}/T_{X,i}$,

where, $T_{o,i}$ is the time that the statically generated version of program P takes to run on the i_{th} input, and $T_{X,i}$ is the time that X takes to run on the i^{th} input.

It is assumed that the assembler only makes minimal changes such that the functions can work properly in the generated executable. It does not alter the performance of the functions.

4. Complexity Analysis

In this section, we investigate the feasibility in finding optimal solutions of the OptMVP. The motivation for this investigation is to understand the computational complexity of the problem, which is important for guiding the direction of efforts: If the problem is NP-complete, efforts may be more worthwhile to be spent on finding effective heuristic algorithms than designing optimal algorithms.

We introduce several assumptions for proving the NP-completeness. Each of the M functions has a copy in every executable. E_0 represents an executable produced by static compilation without FDO. It is assumed that any function version from executable $E_i (1 \le i \le N)$ performs no worse than the corresponding version from E_0 on any input in terms of exclusive time. (Exclusive time refers to the time spent on the function itself without counting the time spent on its callees.)

We use $R^k_{v_{i,j}}$ to denote the reduced exclusive time of the i^{th} function from executable E_j (namely $v_{i,j}$) on the k^{th} input, compared to its corresponding version from executable E_0. The unit of exclusive time is millisecond, and hence $R^k_{v_{i,j}} \in Z^+, 1 \le i \le M, 1 \le j, k \le N$. The space budget B is the allowed code space increase over the code size of E_o.

The corresponding decision problem of OptMVP is given a speedup H, is the overall highest speedup $\sum_{i=1}^N T_{o,i}/T_{X,i} \ge H$? We first prove that the decision problem of OptMVP is NP-complete via the reduction from the PARTITION problem. The formal definition of PARTITION problem is as follows:

Definition 4.1. PARTITION problem
Given A, a set of n positive integers $a_1, a_2, \ldots, a_n$, is there $A' \subseteq A$ such that $\sum_{a_i \in A'} a_i = \frac{1}{2} \sum_{a_i \in A} a_i$?

PARTITION problem has been proved to be NP-complete[18]. We now reduce PARTITION problem to OptMVP. Given an instance of PARTITION problem, we construct an OptMVP problem as follows:

- Let $N = 1$
- For $\forall a_i \in A$, Let $l(v_{i,1}) = R^1_{v_{i,1}} = a_i$
- Let $B = \frac{1}{2} \sum_{v_{i,1} \in C} l(v_{i,1}) = \frac{1}{2} \sum_{v_{i,1} \in C} R^1_{v_{i,1}}$.

The intuitive meaning of this special OptMVP problem is as follows. There is only one input to consider. There are two executables, one generated by static compilation, the other generated by FDO on the profile on that given input. The objective is to pack some parts of the second executable with the code in the first executable to form one "fat" executable, which gives speedup (relative to $T_{0,1}$) over H while keeping the executable size under B. Let Q be the total saved time when a speedup reaches H— that is, $Q = \lceil T_{o,1} - \frac{T_{o,1}}{H} \rceil$. The objective can be then expressed as follows:

Find $S \subseteq C$ such that

$$\sum_{v_{i,1} \in S} l(v_{i,1}) \le B$$
$$\sum_{v_{i,1} \in S} R^1_{v_{i,1}} \ge Q.$$

It can be seen that in this problem, we have $\sum_{v_{i,1} \in S} l(v_{i,1}) = B = Q = \frac{1}{2} \sum_{v_{i,1} \in C} l(v_{i,1})$. Therefore, a solution to the OptMVP problem, S, is also a solution to the original PARTITION problem.

On the other hand, given a PARTITION of A that $A' \subseteq A$ and $\sum_{a_i \in A'} a_i = \frac{1}{2} \sum_{a_i \in A} a_i$, let $A' = S$, We have

$$\sum_{v_{i,1} \in S} l(v_{i,1}) = \frac{1}{2} \sum_{a_i \in A} a_i \le B$$
$$\sum_{v_{i,1} \in S} R^1_{v_{i,1}} = \frac{1}{2} \sum_{a_i \in A} a_i \ge Q$$

So we have shown that a solution to Opt-MVP is also a solution to the PARTITION instance and vice versa. It is easy to see that the reduction can be done in polynomial time—which is bounded by $O(|A|)$. Since PARTITION problem is NP-complete, the decision problem of Opt-MVP is NP-hard. It is obvious that checking whether a potential Opt-MVP solution satisfies the constraints can be done in polynomial time. Hence, the Opt-MVP decision problem is in NP, and hence is NP-complete. We can solve the original OptMVP optimization problem by asking the decision problem in polynomial time, therefore the Opt-MVP is NP-complete.

5. CHoGS Algorithm

This section presents our solution to the OptMVP problem, a heuristic algorithm named Callgraph-based Holistic Greedy Selection (CHoGS). To better understand the design rationale of the CHoGS algorithm, it helps to first examine the complexities for solving the OptMVP problem heuristically.

5.1 Design considerations

Solutions to the OptMVP problem requires answers to the following two fundamental questions:

- Q1: How many versions should each function have in the final executable?
- Q2: What should be those versions?

First Question Answers to the first question depend on the *hotness* of the function and its *input-sensitivity*. Here, the hotness of a function is defined as the ratio between the accumulated exclusive time of the function (i.e., the time consumed by the function itself, excluding its callees) and the overall execution time of the program. The larger the hotness is, the more the function weighs in the whole program. Optimizations of such a function would have a larger potential influence on the overall program performance. But if the function is not sensitive to inputs—that is, all versions of the function perform similarly well, the function still may not deserve having multiple versions.

The complexity is that the hotness of a function itself is often input-sensitive: A function may be hot in one run but cold in another. Moreover, some functions could get completely inlined in some but not all versions of the program executable. The hotness of the function in those versions becomes tricky to determine. So the central challenge for answering the first question is how to address the tightly-coupled relations between hotness and input-sensitivity, and how to handle the side effects of compiler optimizations.

Second Question Answering the second question requires the determination of the quality of a version of a function—for which, neither exclusive time nor inclusive time (e.g., time spent on the function and all its direct or indirect callees) of the function could work directly. Using exclusive time, for example, is subject to the function inlining effects. Consider two versions of the program executable: P_1 and P_2, and their executions on a single input. In P_1, function A calls B, the exclusive time of A is $5s$, and the exclusive time of B is $5s$; in P_2, the call to B is inlined into A and the exclusive time of A (now B is part of it) is $8s$. Even though the first version of A has a shorter exclusive time than the second, it is apparently inferior in terms of its influence on the overall program running time.

It seems that for that particular example, if we use inclusive time of a function, we would get a correct answer. But it is not a general solution. For example, consider two versions of the program executable P_1 and P_2 and their execution on a single input. In both versions, A calls B once. In P_1, A itself takes $5s$, and B itself takes $5s$, while in P_2, A itself takes $3s$, while B takes $9s$. Based on inclusive times, one would select the first version of A, but apparently the second version of A is more efficient—if one selects the second version of A and the first version of B, the overall execution time would turn to $8s$, less than the time of either of the two runs.

Moreover, OptMVP requires the considerations of both performance and code size. Neither inclusive nor exclusive time considers code size. Code size often conflicts with the performance of a function, because compilers tend to do intensive optimization on hot functions. Some optimization, such as function inlining and loop unrolling, could significantly increase code size. The interesting thing is, if a function is also input sensitive, The specific optimization according to one input may become the reason of performance degradation on another input. As we known, direct side effects of increasing code size includes causing more cache miss as well as putting pressure on register allocation. A well optimized version of an input sensitive function implies it may have relatively large code size and can only perform well on very limited inputs and worsen the others, selecting it or not can't be straightforwardly determined when we have code space concern.

Finally, recursive function calls add further complexities. They complicate the calculation of inclusive execution time of a function.

5.2 CHoGS algorithm

To address the complexity we mentioned in the previous subsection, we develop CHoGS, an algorithm that consists of an initial selection and several iterations with each iteration adding one version to the versioning candidate collection—that is, the collection of versions to assemble together into the final program executable—until the total size reach the code space limit.

Features and Algorithm Input CHoGS has several features:

- **Holistic** Its selection of function versions is oriented by the influence that the version may cast on the *whole-program* execution time, resilient to the effects of different inlining or other compiler optimizations.
- **Callgraph-based** It estimates the influence through a bottom-up calculation on the dynamic call graphs of the program. The calculation accommodates the differences in the call graphs of different versions of the program.
- **Greedy** The algorithm each time adds into the version collection the version that, based on the current collection, can maximize the performance of the program per space cost.

Figure 2 outlines the CHoGS algorithm (the function "calProgExecTime" is shown in figure 3). For simplicity of explanation, the description first assumes that there is only one input, and later explains how to handle multiple inputs. We leave the treatment of recursions to Section 5.3, and leave algorithm optimization to section 5.4.

As we mentioned earlier, we have N executables which each is compiled with FDO enabled and uses different profiles. There are M functions in an executable, accordingly, each function has N versions. There are N profiling reports corresponds the time measurements of the N executables' executions on the input. For each profiling report, we use following data structure to store corresponding profiling information. Those data structure is also used in figure 2 and figure 3.

DCG DCG[funcId][versionId] represents the dynamic call graph of a function version, DCG is short for "dynamic call graph". This data structure captures a function version's direct callee and callee's invocation number that called from the function version. For example, DCG[funcId][versionId] = { [callee1, call#], [callee2, call#],. . . }.

invo invo[funcId][versionId] denotes the total number of invocation of a function version.

timeProf timeProf[funcId][versionId] is the time profile of a function version, consisting of two components, [self, children]. The first is the time spent on itself (i.e., exclusive time), and the second is the sum of the inclusive time of all its callees.

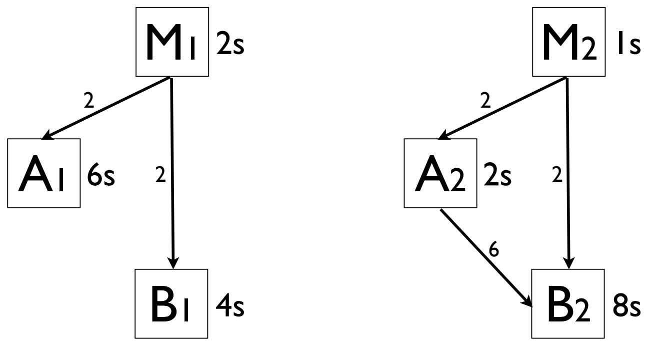
codeSize codeSize[funcId][versionId] records the code size of a function version.

First Step of CHoGS The first step of the CHoGS algorithm creates an initial versioning candidate collection. The initial selection is straightforward, we select the executable which has the minimum execution time, then add all function versions from this executable into the candidate collection.

Second Step of CHoGS In the second step, CHoGS algorithm incrementally adds more function versions into the versioning candidate collection, one version per iteration. In each iteration, each version outside the versioning candidate collection has an opportunity to show its impact to the whole-program execution time.

The estimation of the impact is based on an assumed perfect runtime version selector, R, whose algorithm is shown in figure 3. Given a versioning candidate set, R can select the best version for a function call. Here, being best means the version can maximize the program performance, assuming R applies to all other function calls. The perfect runtime version selector uses a bottom-up method to calculate the inclusive times of function calls. Given a function, the perfect runtime version selector first select versions for its callees from the current candidate versioning set and computes the inclusive times spent on these callees. With the time spent on the function itself and the function's total invocation number, it can then compute its inclusive time per invocation. The algorithm is shown in function *callInclusiveTime* in figure 3, it recursively calls it self and starts to return when meeting a "leaf" function in term of topological position on the call graph.

As mentioned in the previous subsection, different versions of a function may have different dynamic call graphs, due to inter-procedural optimization such as inlining. It makes the direct usage of exclusive time or inclusive time fail, but does not affect the bottom-up estimation by our perfect selector. Figure 1 shows an example to better illustrate how the perfect runtime version selector tolerates the complexity. Suppose we have two executables a and b, their dynamic call graph is shown in 1. Profiling information of each function, such as invocation number and time spent on itself, is known as labeled in the graph. The perfect runtime selector works in a bottom-up fashion. It will select a version for function B first. Since B is an "leaf" function, its inclusive time is itself. The inclusive time per invocation of B_1 and B_2 are $2s$ and $1s$, thus B_2 will be selected. A_1 is a "leaf" function, too, its inclusive time per invocation is 3s. When we compute A_2's inclusive time, the perfect runtime selector will select B_2 when function A callee B, thus the inclusive time of A_2 is $2s + 6s = 8s$ and its inclusive time per invocation is $4s$, which is worse than A_1's. Thus A_1 will be selected by runtime version selector. Finally the inclusive time of M_1 and M_2 are $10s$ and $9s$, M_2 will be selected. As we can see, A_2 spent less time on itself but A_1 is selected by runtime version selector. Selecting A_1 does shorten the execution time of M_2.



(a) executable a (b) executable b

Figure 1: Example of how perfect runtime version selector works(invocation number is on the edge and time spent on function itself is denotes nearby).

With the perfect runtime version selector, at the beginning of each iteration in Step 2 of CHoGS, we can calculate the program's minimum execution time by choosing the best versions from the current versioning candidate collection at each function call. We denote that time as *curExecT*. Then we examine versions outside the versioning candidate collection one by one. When one version is added into the current versioning candidate collection, we recompute the program's execution time, represented as *evalExecT*. To take both performance impact and code size into consideration, we define a term *speedupSizeRatio* to evaluate each version. The definition is as belows:

Definition 5.1.

$$speedupSizeRatio = \frac{curExecT - evalExecT}{evalExecT} \Big/ size$$

Based on the definition, a version that reduces more execution time and has a smaller code size will have a larger value of *speedupSizeRatio*. CHoGS sorts the versions based on their *speedupSizeRatio* in descending order, then selects the version having the largest *speedupSizeRatio* and adds it to the versioning candidate collection. Exceptions happen when the total size of the versioning candidate collection approaches the code space limit, and adding a function with the largest *speedupSizeRatio* exceeds the code space limit. In this case, CHoGS examines each version in the sorted list and adds the first one that meets the space requirement. CHoGS terminates when the code space is filled up or all versions have been added in.

Applying to Multiple Inputs So far we have explained the CHoGS algorithm on one input. Applying CHoGS to multiple inputs needs only two simple extensions.

The first is that in the first step of CHoGS, when selecting the best program executable as the initial content of the versioning collection, one should use the performance of a program executable averaged across all inputs.

The second is that in the second step of CHoGS, one needs to define *speedupSizeRatio* with all inputs considered. If we would like to minimize the total execution time on all inputs, we can first sum up the curExecT and evalExecT on all inputs. If the goal is to maximize the averaged speedup on all inputs, we can compute speedupSizeRatio for each input, then add them together. In either case, the change is just a slight modification to the calculation of *speedupSizeRatio*.

5.3 Addressing recursive calls

Recursive calls are common in programs. Self recursive functions are handled by CHoGS just as normal functions. But mutual recursive calls could impose some obstacles to the CHoGS algorithm. As the perfect selector works bottom up on the call graphs, it first selects the callees' best versions based on the inclusive time per invocation, then updates the caller's inclusive execution time. Recursion blurs the boundary between caller and callee and cause difficulty to the perfect selector. Additionally, it also causes troubles to profiling tool to accurately measure the time spent on callers and callees.

Profiling tools such as gprof[8] treat recursive calls specially. It creates a new entry for mutual recursive call, named with "cycle" followed by a number. From now on we use *cycle* to describe mutual recursive call. The callee of the entry are the members of the cycle.

A *cycle* behaves like a single unit. Table 1 shows an simplified example of a cycle from SPEC program Gap. The first line in the table is the explanation of each column. In the example, cycle1 has two members, *fee* and *foo*. Cycle1's self time is the sum of *fee* and *foo*'s self time($5.79 = 1.33+4.46$) and cycle1's descendents time (i.e., accumulative inclusive time of its callees) is the summation of its member's descendents times ($6.62 = 2.23 + 4.43$). When a function outside cycle 1 *InitGap* called function *foo*, Rather than using the self and descendents of *fee* (1.33 and 2.23), the proportionate time of cycle1 is used ($4.63 = 6.62 \times 120/150$ and $5.3 = 6.62 \times 120/150$) since invocation of *fee* would indirectly causes other members of cycle1 to be invoked.

The CHoGS algorithm addresses recursive calls in a similar manner. Given a versioning candidate collection, for each cycle the CHoGS algorithm will select best version for its members as CHoGS algorithm does to normal function. Then compute cycle's self and descendents time by summing up the self and descendents time of members, respectively, as cycle 1 shown in table 1. When a function outside the cycle calls a member of the cycle, the proportionate time of the cycle's self and descendents time will be used for that function(as *fee* called by InitGap in table 1). When a function outside the cycle calls a member of the cycle, the proportionate time of the corresponding cycle will be computed as time spent on a callee of that function.

```
  // # of functions: M
  // # of versions: N
step1 (initial selection):
  candidateSet = {}
  minExecT = +∞
  minVsn = 0
  for vsnIdx = 1 to N do
    curExecT = timeProf['main'][vsnIdx].self
                   +timeProf['main'][vsnIdx].children
    if ( minExecT > curExecT ) then
      minExecT = curExecT
      minVsn = vsnIdx
    end
  end
    For funcIdx in executable[minVsn] do
      candidateSet.add( [ funcIdx, minVsn ] )
    end
step2 (add):
  progCS = calCodeSize( candidateSet )
  while progCS < limit:
    curExecT = calProgExecTime ( candidateSet )
    spdupSizeRatioList = {}
    for funcIdx = 1 to M do
      for vsnIdx = 1 to N do
        if [ funcIdx, vsnIdx]  not in candidateSet then
          tmpSet = candidateSet
          tmpSet.add( [ funcIdx, vsnIdx ] )
          evalExecT = calProgExecTime ( tmpSet )
          speedup = ( curExecT - evalExecT ) / evalExecT
          spdupSizeRatio = speedup / codeSize[funcIdx][vsnIdx]
          spdupSizeRatioList.add([spdupSizeRatio,funcIdx,vsnIdx])
        end
      end
    end
    // all versions are added
    if spdupSizeRatioList.length() == 0 then
      break
    end
    DescendingSortOnSpdupSizeRatio( spdupSizeRatioList )
    idx = 0
    while idx < spdupSizeRatioList.length() do
      item = spdupSizeRatioList.item( idx )
      tmpSet = candidateSet
      tmpSet.add( [ item.funcIdx, item.vsnIdx ] )
      if calCodeSize( tmpSet ) < limit then
        candidateSet = tmpSet
        break
      end
      idx += 1
    end
    // no function can fit into the left code space
    if idx == spdupSizeRatioList.length() then
      break
    end
  end
  return candidateSet
```

Figure 2: CHoGS algorithm

Table 1: Example of gprof's report of recursive calls (or called cycles)

index	%time	self	descendents	total called+self called/total	parents name children	index
4	41.7	5.79	6.62	150+3650	(1 as a whole)	4
		1.33	2.23	482	fee (cycle 1)	6
		4.46	4.43	790	foo (cycle 1)	14
5	40.7	0	12.1	1	InitGap	5
		4.63	5.3	120/150	fee (cycle 1)	14
6	12	1.33	2.23	482	fee (cycle 1)	6

```
function calProgExecTime( candidateSet )
  [progExecT, selectedVsn] = calInclusiveTime('main',candidateSet)
  return progExecT

function calInclusiveTime( funcId, candidateSet )
  vsnPool = {}
  for vsnIdx in candidateSet[funcId] do
    callerIncT = timeProf[vsnIdx][funcId].self
    // If has callee, updates time spent on callee
    if DCG [funcId][verIdx] $\neq \emptyset$ then
      for callee in DCG[funcId][vsnIdx] do
        [calleeIncT,selectedVsn]=calInclusiveTime(callee, candidateSet)
        // compute time spent as a callee of the caller
        calleeMinPerInvo = calleeIncT / invo[callee][selectedVsn]
        invoFromCaller = DCG[funcId][verIdx].callee.call#
        callerIncT += calleeMinPerInvo * invoFromCaller
      end
    end
    callerPerInvo = callerIncT / invo[caller][vsnIdx]
    vsnPool.add( [ callerPerInvo, callerIncT, vsnIdx] )
  end
  AscendingSortOnCallerPerInvo( vsnPool )
  minCallerIncT = vsnPool[0].callerIncT
  selectedVsn = vsnPool[0].vsnIdx
  return [ minCallerIncT, selectedVsn ]
```

Figure 3: Perfect runtime version selector

The dynamic call graphs of different executables could have different cycles on the same input. This isn't a problem to CHoGS algorithm. It always first selects the best versions for cycle's member functions. During the selection, it treats them as normal functions. It next computes the cycle's self and children time (sum up member functions' self and children time, respectively). Finally update the inclusive time of the functions that call any member of the cycle accordingly.

5.4 Optimizing CHoGS algorithm

The algorithm described in figure 2 shows the basic idea of the CHoGS algorithm. Our implementation optimized it in several ways. The most time consuming part of the algorithm is the perfect version selection shown in figure 3, where the function *calInclusiveTime* recursively calls itself. We have several optimizations on it.

We use dynamic programming to efficiently track the best versions in one invocation of the "calProgExecTime" function. The basic observation is that given a versioning candidate set, the best version of a function is fixed, thus we can return immediately if we keep tracking those best versions. Moreover, for efficient evaluation of versions, we build up an affecting map for each version. A version's affecting map contains all its direct and indirect callers. At the beginning of one iteration in the second step, we keep a list that records the inclusive time of all functions returned by *calProgExecTime*; when we evaluate a version, we remove the functions on its affecting list and then use the map to avoid computations for those unaffected functions.

We also implement a benefit estimator to filter out the versions, adding which has no impact on the version selection. First of all, we filter out versions whose accumulative self time on all inputs are negligible. Second, on a given input to the program, we maintain a list that records the minimum inclusive time per invocation on that input for all functions in the current candidate set. When we evaluate a version by temporarily adding it in, we first compute its time spent on its callees by using their minimum inclusive time per invocation from the list. With the time spent on the version itself and its total number of invocations on the input, we can compute the version's inclusive time per invocation. If the inclusive time per invocation of this version is longer than the existing inclusive time per invocation, this version is worse than what the current candidate set already has, and hence will be ignored. If we run on multiple inputs, a version will be ignored if none of its inclusive time per invocation on any input is better than existing inclusive time per invocation. If a version's inclusive time per invocation is better on some inputs, we only invoke the *calProgExecTime* function to measure their impact to executions on those inputs.

The time complexity of CHoGS algorithm is $O(MN)^2$, the worst case is all versions are hot and the extra space is enough to add all of them. However, in practice the performance of CHoGS is fast. The reason is in most program a large portion of execution time is on several functions. We actually only need to evaluate and select versions for those hot functions.

6. Implementation

We used IBM XLC compiler to produce executables with FDO enabled. We also modified XLC's optimizer, Toronto Portable Optimizer(TPO) to dump out the estimated binary code size of each function before the optimizer passes intermediate code to XLC's backend to generate final executable.

We take advantage of gprof[8] to gather fine-grained time profiling for all executables on each input. Gprof requires a program to be compiled with a special option "-pg". After an execution of the resulting executable, a file named "gmon.out" is produced. A command *gprof <executable name> gmon.out* creates a fine-grained time profiling report for that execution, which includes a function's invocation frequency, dynamic call graph and time spent on itself and children, respectively.

For each function, gprof provides statistical time profiling information, which does not show the variation of time spent at different call sites. As we mentioned in section 2, this study focuses on input sensitivity for versioning rather than calling contexts. We assume through an execution of a specific input, one selected version of a function is used for all call sites. Therefore, the gprof report is sufficient for this study.

7. Evaluation

In this section, we verify the effectiveness of the CHoGS algorithm. We design experiments to answer the following questions:

- What is the potential of doing function versioning for a program? That is, given all function versions, how much speedup could it achieve?
- Given different code space budget, how well can the CHoGS perform comparing with the potential?
- Is the design of CHoGS necessary? Could a simple hotness-based method already suffice?

For the third question, an intuitive hotness-based approach is implemented to compare with CHoGS. The initial versioning set of the two methods are function versions from the executable which has the minimum accumulative execution time on all inputs. The difference is, CHoGS adds other versions based on its algorithm, while the hotness-based approach first sorts functions according to the accumulative exclusive time of themselves on all inputs in descending order, then add all available versions of those functions which located at the top of the sorted list, until the code space limit is reached.

7.1 Methodology

We examined 11 programs from SPEC CPU 2000 and CPU 2006, ranging from utility programs to database, scientific computing program, programming language interpreter and compiler, circuit routing and game. We select these programs for they are sensitive to FDO (i.e., their FDO version gives more than 5% speedup over their static compiled counterparts). Each program comes with some inputs by default; we collected more inputs to increase the input diversity. In our experiment, each program has 10 inputs and 10 executables are produced by the FDO with each based on the profile collected on one input. Table 2 summarizes characteristics of the 11 programs and the range of execution times for the differences in inputs.

Even though the space efficiency is a more prominent concern on portable and embedded systems, our current framework (including the XLC compiler) is not yet ready to run on such systems. To prove the concept, we conduct our experiments on an IBM Power 8 machine. Technically speaking, it may be possible to assemble code of functions generated by FDO on different inputs into one single executable. However, the current compiler does not support it. For proof of concept, we use a trace-based approach. By profiling repeated executions of each of the benchmarks, we collect the average value of $T_{o,i}$ and $T_{e,i}$ for every function, where, $T_{o,i}$ is the average per-invocation execution time of that function when the statically generated executable runs on the ith input, and $T_{e,i}$ is the average per-invocation execution time of that function when the executable e (produced by the FDO) runs on the ith input. Through the same process, we also record the number of invocations of each function in every program execution. With all the information, the execution time of an arbitrary executable can be calculated. The calculated time may have some disparity from the time that the really assembled executable could take. But none the less, the relative performance offers some evidence on the effectiveness of the introduced space-efficient multiversioning. Moreover, such a trace-based evaluation simplifies the assessment of the full potential of the technique by allowing the circumvention of the difficulty in implementing the oracle runtime version selector—the construction of which is out of the scope of this work.

7.2 Results

We report the experimental results in three parts. The first part examines the full potential of versioning on individual input and on average, the second part compares CHoGS with the hotness-based approach, and the last part verifies the effectiveness of CHoGS through leave-one-out cross-validation.

Figure 4 presents the potential speedup of doing function versioning of 11 programs, the base is the execution time from the executable which among all 10 executables has the minimum accumulative execution time on all inputs. In the

Table 2: Characteristics of Benchmarks

Program	Description	# of inputs	# of functions	Execution time range
bzip2	compression utility	10	65	20.16s - 82.73s
crafty	chess game	10	95	5.45s - 250.81s
gap	group theory, interpreter	10	837	1.09s - 60.43s
gcc	compiler	10	4438	3.53s - 117.52s
gzip	compression utility	10	61	6.9s - 24.31s
milc	quantum computing	10	334	1.84s - 22.74s
parser	word processing	10	159	1.34s - 25.11s
perlbench	perl interpreter	10	1526	2.77s - 89.32s
sjeng	chess game	10	276	2.43s - 75.52s
vortex	object-oriented database	10	1050	1.49s - 25.63s
vpr	FPGA circuit placement and routing	10	192	0.84s - 20.67s

graph, the inputs for each program are sorted by execution time in ascending order.

We observed 8 out of 11 programs exhibit substantial speedup, suggesting that they are input sensitive. Bzip2, gzip and vpr are not very sensitive to inputs. It is possible that the inputs we used for them are not diverse enough, even though different file formats, including text, audio, video and graphs are included for the two compression utility programs, bzip2 and gzip.

Another observation is that programs with a larger number of functions are more likely to have the input sensitivity problem. Gcc, perlbench, vortex and gap each have more than 800 functions; they all show strong input sensitivity. One observation is that on those programs, intensive interprocedural optimizations such as inlining are conducted by the compiler. By checking compilation logs we found gcc, perlbench, vortex and gap respectively have 12597, 2030, 1904 and 901 functions inlined in a typical compilation. Inlining is one of most beneficial transformations by FDO and inlining decisions are often influenced by profiles significantly. Intensive inlining according to one profile can easily magnify the disadvantage of inlining and gain no benefit on other inputs. Therefore, doing function versioning is a necessary optimization to alleviate this problem with the presence of FDO.

Figure 5 illustrates the comparison of CHoGS and the hotness-based approach. Each program is given a spectrum of amount of extra code space. For example, 0.1 denotes a size of $0.1 \times avgsz$, where $avgsz$ is the average size of the 10 executables of a program. To be consistent with Figure 4, the base is the execution time from executable which has minimum accumulative execution time on all inputs. The speedup is the averaged speedup of all inputs. We can see the CHoGS algorithm approaches the potential speedup much faster than hotness-based method. (Note the axis is in an exponential scale.) For all programs except sjeng, we can see CHoGS achieves more than 87% of the potential average speedup with only 10% extra code space, and over 96% with 40% extra code space. Sjeng achieves 81% and 88% speedups with 20% and 40% extra code space, separately. As a comparison, the hotness-based approach performs much worse than CHoGS at when only a small amount of extra code space is allowed. It requires much more extra space to reach the potential speedup, typically $3.2X$, while gcc and sjeng need even more extra space.

The limitation of the hotness-based approach is that it selects versions that are hot in terms of time spent on itself, but it fails to capture the input sensitivity of the versions. An extreme case is crafty. We manually checked its log files that record which versions were added by the hotness-based approach. We find that in crafty, there is a function called "MakeMove" that takes a considerable amount of time by itself, but performs stably across inputs. The size of "MakeMove" function is large. After adding it into the candidate set, the hotness-based method gains little but spends a lot of space budget. In contrast, CHoGS identifies this function isn't beneficial at a very early stage, and uses the space for other smaller and more profitable functions.

Figure 6 shows the result of leave-one-out cross-validation for CHoGS when 0.4X extra code space is given. Every time one input is taken out for testing purpose and profiles from the other 9 inputs are applied to generate 9 executables. CHoGS selects function versions from the 9 executables to run on the testing input, and the execution time is compared with the shortest execution time from the 9 executables on the testing input. The process is repeated 10 times with a different input taken out for testing each time. The boxplot depicts the result. Similar to Figures 4 and 5, 8 out of the 11 programs display notable speedups, with geometric medians ranging from 1.10 (sjeng) to 1.51 (gcc).

The CHoGS algorithm takes only a short time to run. It finishes within seconds for most SPEC programs. On *Gcc*, *perlbench* and *gap*, it takes slightly longer because they have more function versions to evaluate. On the largest program *Gcc*, it takes 5 mins when it considers function versions

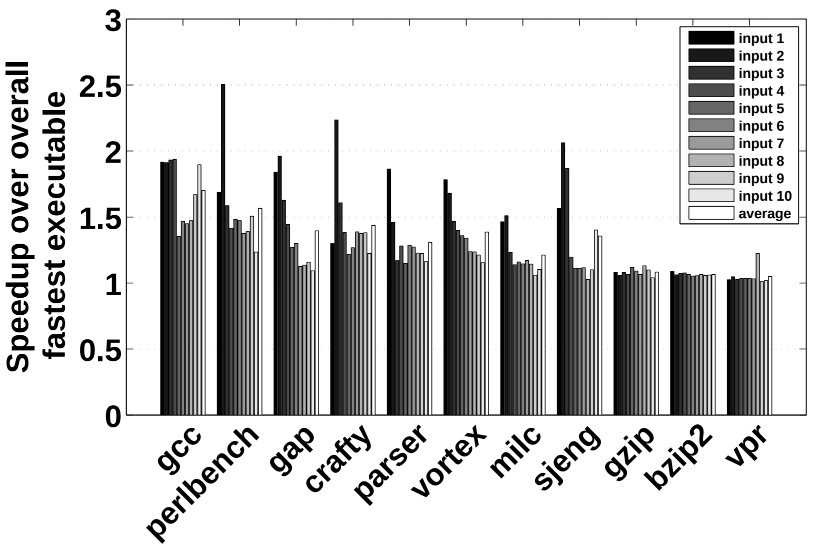


Figure 4: potential speedup of doing function versioning, inputs are sorted by execution time in ascending order.

whose accumulative time on all inputs are larger than 0.1 seconds.

Overall, the experiments confirm that the CHoGS algorithm can effectively select the appropriate versions to include to overcome input sensitivity problems in a space-efficient manner.

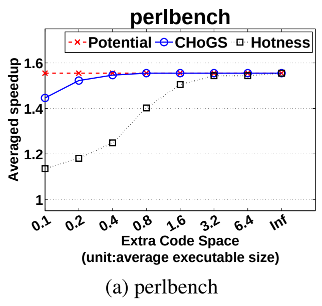
8. Related Work

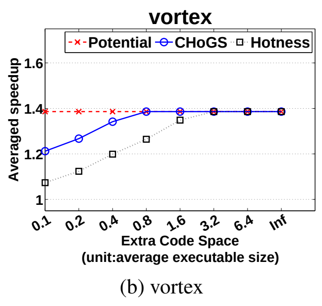
This work makes following main contributions: giving a formal definition of OptMVP and uncovering its computational complexity; proposing an effective approximation algorithm that is able to approximate the potential speedup with very low extra code space requirement. To our best knowledge, the OptMVP problem has not been systematically explored before.

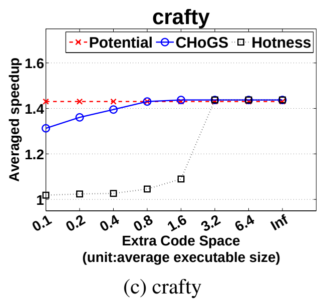
The most closely relevant work we found includes the studies by Chuang and his colleagues [6] and others [5, 25], which conducts online code version selection based on runtime profiles. Our current study differs from the previous work in several major aspects. First, their empirical study does not reveal the complexity of version selection problem or consider code size. Second, the previous work generates versions with different optimizations in aggressiveness adjustment of loop scheduling, load prefetch, and control speculation, respectively. Our versions are generated with all optimizations contained in the IBM commercial compiler, including inlining and other inter-procedural optimizations. CHoGS answers the theoretical question on the full potential speedup of doing multi-versioning, and provides a practical solution to save code size while maintaining the benefits brought by multi-versioning.

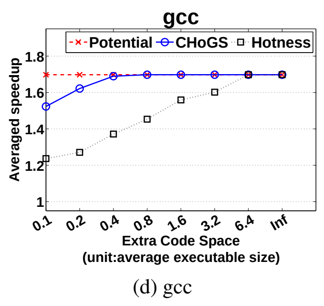
There are a body of work that uses input features to guide dynamic program optimizations, such as the computation offloading by Wang and Li [26], the adaptive algorithm selection from Rauchwerger's group [20] and Li and others [13]. Shen and Mao proposed an input characterization language that converts inputs into attribute vectors and uses machine learning technique to remove redundancies [17]. Berube and his colleagues provide a compiler-centric clustering approach to reduce the number of representative workloads [1]. Tian and others [22] have proposed an input-centric program dynamic optimization framework, which offers a systematic way to include program inputs into the dynamic optimization process. They later [24] eliminated the needs for offline training, making the framework deployable in a way completely transparent to users. Jung and others have considered inputs when selecting the appropriate data structures to use [11]. There are some other studies on cross-input adaptive optimizations, in library constructions [3, 7, 9, 15, 27], GPU program optimizations [14, 16], and so on. A recent study [29] introduces the concept of *principled speculative parallelization*, a rigorous way to exploit the input features of Finite State Machines for parallelizing such challenging, inherent sequential applications. These studies typically concentrate on runtime tuning and selection of some configuration parameters. Some of them involve code version selection, but without exploring the effects of code size increase.

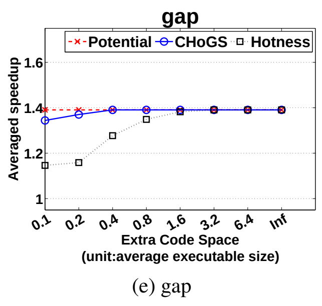
Some other studies have investigated the correlations among program components for static programming languages. The correlations have the potential to serve as heuristics for selecting the suitable code versions at runtime. Jiang and his colleagues [10] have conducted a systematic study about the statistical relations among the different components of a program. They found that strong correlations commonly exist among loops in a program, which indicates the possibility to predict the behaviors of a loop (e.g., number of iterations) through the behaviors of another loop. The

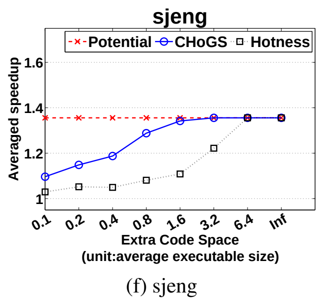

(a) perlbench

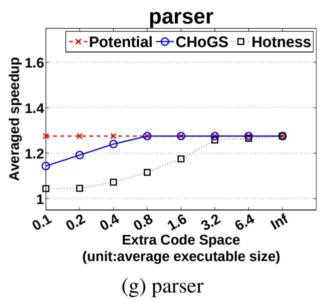

(b) vortex

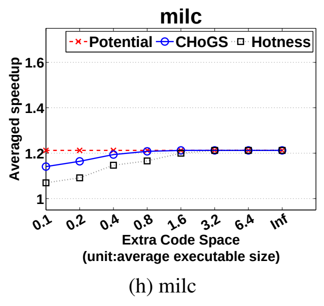

(c) crafty

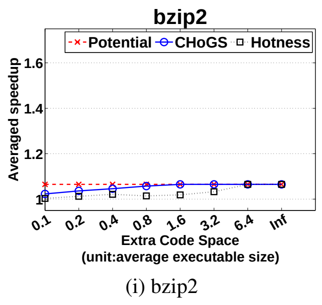

(d) gcc

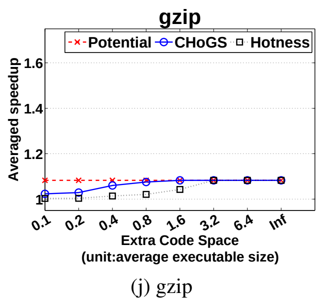

(e) gap


(f) sjeng


(g) parser


(h) milc


(i) bzip2


(j) gzip

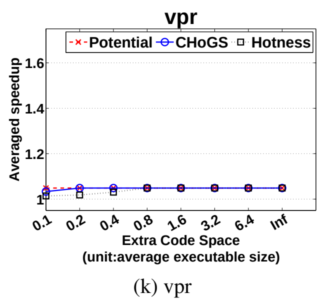

(k) vpr

Figure 5: Comparison of CHoGS and hotness-based approach under different extra code space, x axis is scaled exponentially.

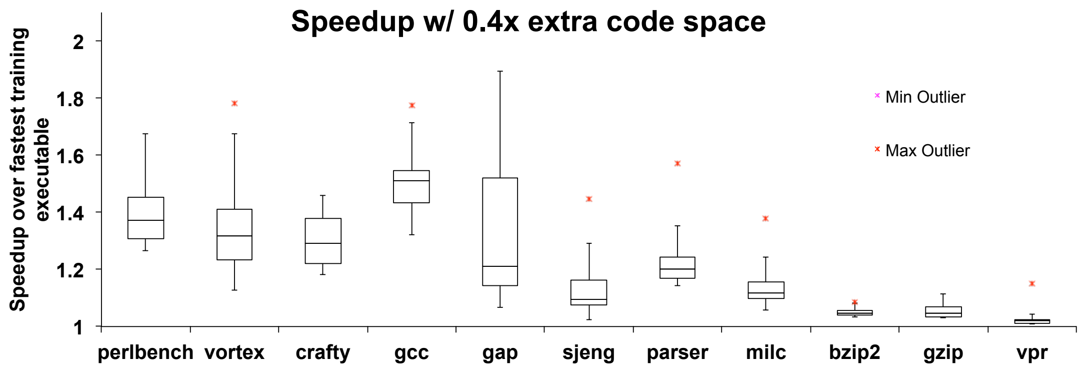


Figure 6: Result of leave-one-out cross-validation given 0.4X extra code space(unit:average executable size)

predictability means the potential for leveraging statistical correlations for guiding runtime code version selection. The authors introduced the notion of *seminal behaviors* accordingly. Wu and others further show that the kind of statistical correlations also exist among the sequences of invocations of different loops [28]. They use such sequences of loops within a function as the signature of that function, which they demonstrate useful for guiding dynamic version selection [28]. These studies are complementary to this current work in that the techniques in them could serve as practical approximation of the dynamic oracle for version selection assumed in this study.

9. Conclusion

In the paper we propose an systematic study on the optimal selective function multi-versioning problem (OptMVP problem). We formally define the OptMVP problem and prove its computational complexity to be NP-complete via a reduction from the well-known NP-complete problem PARTITION. Driven by the NP-completeness of the OptMVP, we proposed CHoGS, a simple but effective approximation algorithm for OptMVP. We evaluate the effectiveness of the CHoGS through 11 SPEC programs. For most of the programs that show notable potential speedup through multi-versioning, CHoGS finds a set of versioning candidates which can yield 96% of the full potential speedup within 40% extra code space cost. It outperforms a traditional hotness-based approach by 4X to 8X in terms of space efficiency (when space budget is 10%). The result suggest that CHoGS is a promising solution to enable space-efficient multi-versioning optimization, which is especially important for mobile and embedded systems.

Acknowledgment

We thank the anonymous reviewers for the helpful comments. This material is based upon work supported by the National Science Foundation under Grant No. 1320796 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] P. Berube and J. N. Amaral. Benchmark design for robust profile-directed optimization. In *Standard Performance Evaluation Corporation (SPEC) Workshop*, 2007.

[2] P. Berube and J. N. Amaral. Benchmark design for robust prole-directed optimization. In *Standard Performance Evaluation Corporation (SPEC) Workshop*, 2007.

[3] J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *Proceedings of the ACM International Conference on Supercomputing*, pages 340–347, 1997.

[4] D. Chen, N. Vachharajani, R. Hundt, S. wei Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. Taming hardware event samples for fdo compilation. In *International Symposium on Code Generation and Optimization (CGO)*, Toronto, Ontario, Canada, 2010.

[5] B. Childers, J. W. Davidson, and M. L. Soffa. Continuous compilation: A new approach to aggressive and adaptive code transformation. In *International Parallel and Distributed Processing Symposium(IPDPS)*, Nice, France, 2003.

[6] P. fei Chuang, H. Chen, G. F. Hoflehner, D. M. Lavery, and W. chung Hsu. Dynamic prole driven code version selection. In *the 11th Annual Workshop on the Interaction between Compilers and Computer Architecture*, 2007.

[7] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[8] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *SIGPLAN symposium on Compiler construction*, pages 120–126, 1982.

[9] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *Int. J. High Perform. Comput. Appl.*, 18(1):135–158, 2004. ISSN 1094-3420. .

[10] Y. Jiang, E. Zhang, K. Tian, F. Mao, M. Gethers, and X. Shen. Exploiting statistical correlations for proactive prediction of program behaviors. In *International Symposium on Code Generation and Optimization (CGO)*, Toronto, Ontario, Canada, 2010.

[11] C. Jung, S. Rus, B. P. Railing, N. Clark, and S. Pande. Brainy: effective selection of data structures. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 86–97, New York, NY, USA, 2011. ACM.

[12] J. Lau, M. Arnold, M. Hind, and B. Calder. Online performance auditing: Using hot optimizations without getting burned. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, Ottawa, Canada, 2006.

[13] X. Li, M. J. Garzarn, and D. Padua. A dynamically tuned sorting library. In *International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, California, 2004.

[14] Y. Liu, E. Z. Zhang, and X. Shen. A cross-input adaptive framework for gpu programs optimization. In *Proceedings of International Parallel and Distribute Processing Symposium (IPDPS)*, pages 1–10, 2009.

[15] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo. SPIRAL: code generation for DSP transforms. *Proceedings of the IEEE*, 93(2): 232–275, 2005.

[16] M. Samadi, A. Hormati, M. Mehrara, J. Lee, and S. Mahlke. Adaptive input-aware compilation for graphics engines. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 13–22, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1205-9. . URL `http://doi.acm.org/10.1145/2254064.2254067`.

[17] X. Shen and F. Mao. Modeling relations between inputs and dynamic behavior for general programs. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 2007.

[18] M. Sipser. *Introduction to the Theory of Computation 2nd edition*. Cengage Learning, Boston, Massachusetts, 1997.

[19] M. D. Smith. Overcoming the challenges to feedback-directed optimization. In *the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization(Dynamo)*, Boston, USA, 2000.

[20] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A framework for adaptive algorithm selection in stapl. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, Chicago, Illinois, USA, 2005.

[21] K. Tian, Y. Jiang, E. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *ACM International Conference on Systems, Programming, Languages and Applications (OOPSLA)*, Reno/Tahoe, Nevada, USA, 2010.

[22] K. Tian, Y. Jiang, E. Zhang, and X. Shen. An input-centric paradigm for program dynamic optimizations. In *ACM International Conference on Systems, Programming, Languages and Applications (OOPSLA)*, Reno/Tahoe, Nevada, USA, 2010.

[23] K. Tian, E. Zhang, and X. Shen. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *ACM International Conference on Systems, Programming, Languages and Applications (OOPSLA)*, Portland, Oregon, USA, 2011.

[24] K. Tian, E. Zhang, and X. Shen. A step towards transparent integration of input-consciousness into dynamic program optimizations. In *ACM International Conference on Systems, Programming, Languages and Applications (OOPSLA)*, Portland, Oregon, USA, 2011.

[25] M. J. Voss and R. Eigenmann. High-level adaptive program optimization with adapt. In *ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, Snowbird, Utah, USA, 2001.

[26] C. Wang and Z. Li. Parametric analysis for adaptive computation ofoading. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, Washington, DC, USA, 2004.

[27] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[28] B. Wu, Z. Zhao, X. Shen, Y. Jiang, Y. Gao, and R. Silvera. Exploiting inter-sequence correlations for program behavior prediction. In *ACM International Conference on Systems, Programming, Languages and Applications (OOPSLA)*, Tucson, Arizona, USA, 2012.

[29] Z. Zhao, B. Wu, and X. Shen. Challenging the "embarrassingly sequential": Parallelizing finite state machine-based computations through principled speculation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 543–558, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. . URL `http://doi.acm.org/10.1145/2541940.2541989`.