ProfMig: A Framework for Flexible Migration of Program Profiles Across Software Versions

Mingzhou Zhou, Bo Wu, Yufei Ding, and Xipeng Shen

Computer Science Department
The College of William and Mary, Williamsburg, VA, USA
{mzhou,bwu,yding,xshen}@cs.wm.edu

Abstract

Offline program profiling is costly, especially when software update is frequent. In this paper, we initiate a systematic exploration in *cross-version program profile migration*, which tries to effectively reuse the valid part of the behavior profiles of an old version of a software for a new version. We explore the effects imposed on profile reusability by the various factors in program behaviors, profile formats, and impact analysis, and introduce *ProfMig*, a framework for flexible migrations of various profiles. We demonstrate the effectiveness of the techniques on migrating loop trip-count profiles and dynamic call graphs. The migration saves significant (48-67% on average) profiling time with less than 10% accuracy compromised for most programs.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processorsoptimization,compilers

General Terms Languages, Performance

Keywords Profile-Driven Optimizations, Profile Migration, Software Update, Change Impact Analysis

1. Introduction

Offline program profiling is important for revealing program dynamic behaviors and guiding program analysis, optimization, and refactoring. It instruments a program with some monitoring instructions so that when the program runs, it produces some profiles to capture some dynamic behaviors of the software. However, the profiling time overhead can be as significant as a factor of hundreds of the original execution time [17]. Moreover, for software with a variety of inputs, the profiling has to run on many inputs to get a comprehensive view of the program.

The problem worsens when software update is taken into consideration. Till now the common practice has been to reprofile from scratch upon a software update. Since modern software gets update as frequently as once every one or two weeks, the discarding-recollecting scheme results in either the usage of a huge amount of computing power for reprofiling or the difficulty for conducting profile-based analysis or optimizations.

In this work, we introduce the concept of *cross-version program behavior profile migration* to alleviate the problem. The basic observation is that an update to a program may not affect every part of the program. It is hence likely that some part of the profile remains valid despite the software update. The goal of profile migration is to build up the profiles of the dynamic behaviors of a new version of a software by effectively reusing part of the old version's profiles and selectively reprofiling the new version. We refer to the process as *profile migration* for brevity.

Although the idea seems natural, the idea has been only preliminarily explored[1]. In this work, we offers a systematic study in this important direction, and provide *ProfMig*, a framework that supports the migration of a variety of program profiles in a flexible, extensible manner.

Specifically, this study consists of five components. First, we explore the relations between profile migration and change impact analysis. *Change impact analysis* refers to software analysis techniques that try to identify the potential consequences of a code change to a program. It has been widely used in program testing [13–15], but not for profile migration. In this study, we explore its role in this new task, examine the applicability of various change impact analysis techniques for the migration of a variety of profiles, and investigate the benefits of a hybrid change impact analysis that adapts to given migration problems.

Second, we analyze the effects of various factors of program behaviors and profile formats on the design of a profile migration system. We reveal the multi-fold complexi-

[1] There is some software on the market claiming to do profile migration, but the data they migrate are user settings rather than program dynamic behaviors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CGO '13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

ties caused by the granularity of the behavior units, the order and nesting properties of the behavior. We show that being oblivious to profile migrations, the default formats of many program profiles are not amenable for migration for either having a high level of abstraction or missing some important meta data.

Third, after analyzing the various complexities, we propose a simple norm of identity and a hierarchical organization of the identities to enable a uniform treatment to a variety of profiles. They provide a representation that unifies the multiple profile migration steps into a cohesive process, simplifying the coordination of the different steps, and enabling easy adjustment of migration granularity.

Fourth, we develop *ProfMig*, a three-stage framework for automatic profile migration with the flexibility for users' customization. It supports migration of a variety of program profiles. By integrating together the techniques developed in this study, ProfMig features flexible adjustment of migration granularity, automatic selection of change impact analysis techniques, and an open design that allows the incorporation of various profiles and profiling systems.

Finally, we introduce a set of metrics, and assess the profile migration system on two types of behaviors, loop trip-counts and dynamic call graphs. *ProfMig* saves 48-67% average (up to 100%) profiling time with less than 10% accuracy compromised for most programs, demonstrating the effectiveness of the proposed techniques for enabling the migration of different types of profiles.

As a seminal study in program profile migration frameworks, this work opens up some new opportunities for lowering the barriers for applying profiling for program analysis and optimizations, and also points out some future directions.

We organize the rest of the paper as follows. Section 2 offers a formal definition of program profile migration and the scope of this study, Section 3 analyzes the complexities for enabling profile migration, Section 4 presents the solutions, including a description of the *ProfMig* framework, Section 5 explains five metrics useful for assessing profile migration, Section 6 reports experimental results, Section 8 discusses the related work, and Section 9 concludes the paper.

2. Concept and Scope

As a problem that has not been systematically investigated, cross-version profile migration deserves a formal definition. Our definition is as follows:

DEFINITION 1. Cross-version program behavior profile migration *is a process that builds up a collection of profiles for a new version of a program by capitalizing profiles of its old versions and selectively profiling its new version.*

In this particular work, we assume that there are no changes in the underlying platforms. Complexities brought by platform changes are orthogonal to software update-caused profile migration: Even when there is no software update, the changes in underlying platforms may also trigger changes in the program behaviors and hence the need for profile update. This paper concentrates on the issues related with version update.

Among the various behaviors of a program, some are closely related with hardware such that whether an entry in their profiles is affected by a code change depends on not only the program code but also the hardware. An example is cache miss. For a load instruction, even if it has no data or control dependences on any changed code, its cache misses can still be affected by the version update. For instance, if after the software update, the instructions right before it bring much more new data into cache than in the earlier version, a cache hit at that instruction may turn into a cache miss in the new version's execution. Certain hardware modeling is necessary to fully capture the effects of a code change on the profiles of this kind of behaviors. In this study, we concentrate on program-level behaviors that do not have such strong hardware dependence so that we can focus on the core issues in profile migration.

3. Complexities

Before designing a migration system, it is necessary to understand the various factors affecting profile migration and the complexities they create. The main sources of complexity are the large variety of program behaviors, their profile formats, and the analysis for finding the scope of code changes.

3.1 Complexities from Program Behaviors

Among the many properties of a program, the following two are closely related with profile migration.

Behavior Unit Behavior unit refers to the level of program constructs at which the profile is collected. This property determines the smallest granularity for profile migration. For instance, in a profile of function returning values, each behavior corresponds to a function, and the unit of migration can only be a function or some constructs beyond. As the units may differ across program behaviors, it is important to equip a migration system with the capability to adapt to migration granularity at a range of levels.

Order and Nesting For some profiles, the collected behaviors form a sequence, and the order of the behaviors in the sequence is critical. An example is a function call sequence. It records the trace of function invocations in a run, useful for identifying hot streams of function calls and dynamic optimizations. The order of two function calls in the sequence is important for that purpose. We call such profiles *order-sensitive profiles* and others *order-oblivious profiles*.

Order-sensitive profiles can be further classified into *isolated* or *nested* ones depending on whether the behavior sequence of one construct may be nested in the behavior sequence of another construct. A function call sequence is a

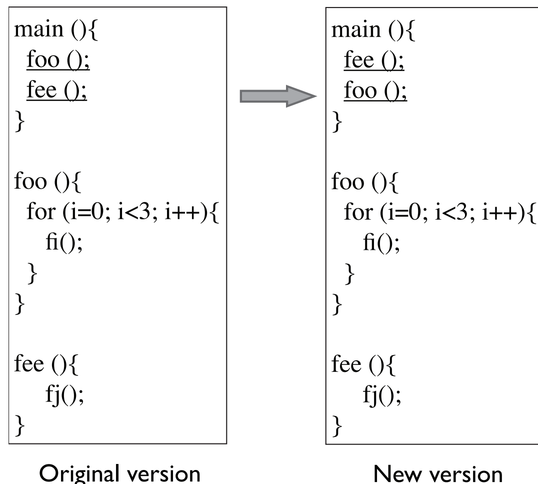

Figure 1. Two versions of an example program.

type of nested profile. Consider the original program in Figure 1. The call sequence of the original program is "main foo fi fi fi fee fj", where the call sequences within foo and fee are embedded in the call sequence in $main$.

Order and nesting have some important implications to profile migration. Without treating them correctly, migration may fall into some pitfalls. For the example in Figure 1, assume there is no data or control flow dependences between foo and fee and the only code change of the new version is the switch of the two underlined function calls. A migration that is oblivious to the order and nesting of the profile may recollect only the invocation order of foo and fee and use that to replace the two corresponding items in the old profile, producing an erroneous profile "main fee fi fi fi foo fj".

3.2 Complexities from Profile Formats

In addition to the behavior properties, the format of the profiling output is also important for profile migration, in mainly two aspects.

Level of Abstraction The first is the level of abstraction of the profiling results. Some profilers process the collected behaviors before outputting the extracted high-level information to the file. This kind of profile is difficult to migrate due to the interplays among program elements that the abstraction process exploits. An example is a hot function profiler that collects the calling frequencies of all functions but only outputs the K most frequently called functions: Whether a function should be output to a profile depends on the calling frequencies of both that function and all other functions. So even if a function does not have dependences on any functions that are modified, its appearance in the profile may still be affected by the version update.

Meta Data The second aspect of profile format is the presence of meta data in the profile for matching the profile content with program constructs. Most profilers, when outputting the collected data, also output the identities of the corresponding constructs. But being migration-oblivious, the output meta data may not suite the needs of profile migration well. A loop trip-count profile, for instance, usually contains the trip-counts as well as the IDs of the corresponding loops. The ID of a loop may be a global serial number the compiler assigns to all the loops in the program. In that case, profile migration may be applicable at loop level but not easy to apply at function level as without extra compiler support, it is hard to map the loops to their enclosing functions just from the profiles. If the ID uses the concatenation of the names of the function and file containing that loops, along with the line number of the loop, the profile migration at function level would become much easier than before. Migration at a level higher than the unit level sometimes provides some practical advantages: The profile at a low level may be difficult for migration due to strong interplays among profile entries or limitations of the change impact analysis. In general, what meta data profilers output determines the applicability and flexibility of the profile migration.

3.3 Complexities from Impact Analysis

An important step in profile migration is to find out the impact of code changes in the new version of software. The basic technique for this purpose is called *change impact analysis* [1]. There have been lots of studies in it, but mainly for software testing. For profile migration, the meaning of getting affected by a code change is slightly different from that in testing. We say that the impact of a code change covers a program construct if the change alters the behavior of the construct such that the profile of the construct shows disparity from its profile in the old version (assuming same inputs and running environment are used.)

Effectively applying impact analysis for this new purpose faces some complexities. The first complexity is that for profile migration, the answer to whether a code change covers a piece of code is not absolute. It depends on both the type of the profile and the program context. In the example shown in Figure 1, the switch of the two underlined statements does not affect the calling frequencies of the two functions, but apparently affects the function invocation sequence. However, for the same example but in a different context, the statement switch may affect the calling frequency of the two functions as well. For instance, if the upper bound of the loop in "foo()" is not 3 but a variable "N" that is modified in "fj()", the frequencies of "fi()" would be affected by the statement switching. So to find out impact scopes precisely, the design of a migration system must consider both the context and the type of the profile.

The second complexity comes from the various granularity, cost, and applicability of existing impact analysis techniques. Existing techniques show a large variety. Some operate at a function level by working on static or dynamic call graphs [1, 15], some at a statement level through static or dynamic program slicing [18], some explore object rela-

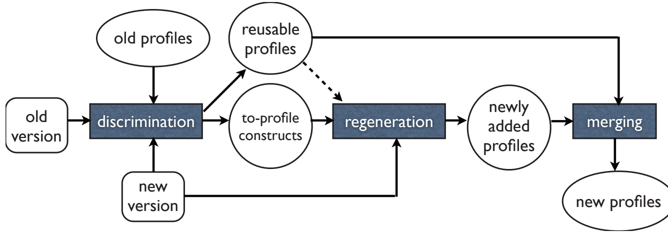

Figure 2. Three steps of profile migration.

tion diagrams [9], some use whole path profiling [13], and some combine different techniques [14]. As they are usually designed for a particular use other than profile migration, none of them consistently excels all others (as confirmed by empirical studies shown in Section 6.) Which one to use depends on the type of profiles, the significance of code changes, and the desirable level of cost and accuracy. The design of a migration system must handle such differences in an adaptive manner.

4. Solutions to the Complexities

Among the complexities listed earlier, some need just a reasonable design consideration, while others—including those in impact analysis and those caused by the variety in behavior units and profile formats—require some systematic support. In this section, we first describe our solutions to the latter category by presenting a norm of identity and *ProfMig*, a general framework for profile migrations. We then describe how the design of *ProfMig* addresses the other complexities, with some insights and guidelines highlighted.

4.1 Overview of *ProfMig*

We first give a high-level overview of ProfMig. ProfMig is a three-stage framework for profile migration, as illustrated in Figure 2. The first stage, *discrimination*, separates the part of old profiles that are still valid for the new version from the others. The second stage, *regeneration*, collects the parts of the profiles that need an update. The third stage, *merging*, combines the valid part of the old profiles with the newly collected part to form the new profiles for the new version.

4.2 Norm of Identity and Hierarchy

Before describing the modules that implement each of the stages, we first present a norm of identity and its hierarchical organization we use as the underlying representation of the target program[2] for all the modules. As the last section mentions, the variety of program behaviors and profile formats demands adaptivity from a migration system. The proposed norm of identity and organization are simple, but meet such a need.

In this norm, for a given program, every construct is identified by a three-element tuple, *(level, file ID, line number)*.

[2] This paper focuses on the cases when the source or byte code of the target program is available.

```
"file_f1"

1:  main (){
2:    ...
3:    foo();
4:    for (i=...)
5:      fee();
6:  }
7:
8:  foo (){
9:    ...}
```

```
"file_f2"

1:  fee (){
2:    for (j=...){
3:      ...
4:    }
5:    for (j=...){
6:      ...
7:    }
8:  }
```

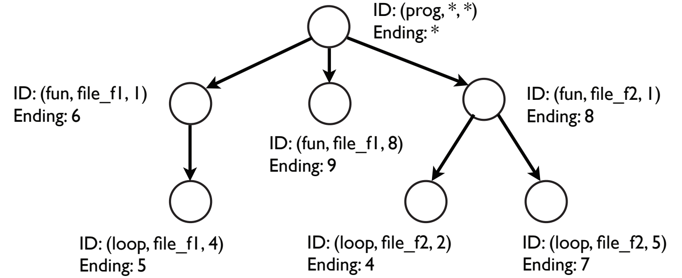

Figure 3. An example construct hierarchy with universal IDs used (for clarity, the IDs are written as strings; they are actually integers obtained through hashing.)

The *level* element indicates the level of the corresponding construct. The current framework supports five levels: statement, loop, function, class, program. New levels can be added through an interface. The second element in the tuple is the ID of the file containing the construct. We use an integer for the ID by hashing file paths and names. The third element is the line number of the first line of the definition of the construct. We call this norm *universal ID*. It provides a uniform, simple way to identify all levels of constructs. It can be further hashed into an integer for conciseness.

With the identity norm, ProfMig builds up a construct hierarchy for a target program. The root node stands for the whole program. Every node in the hierarchy corresponds to one construct entity in the program. The fields of a node record the construct entity's universal ID, ending line number (i.e., the final line of the definition of the construct in the file), and a list of pointers pointing to its children nodes. The children nodes correspond to the constructs the parent node's corresponding construct contains. Statement-level nodes are not explicitly shown in the hierarchy: As the universal ID of the statement contains its line number and the starting and ending line numbers are recorded in each node, its parent node can be easily determined. Figure 3 shows an example of the hierarchy. The nodes at the second level from the top correspond to the three functions defined in the two files; the nodes at the third level correspond to the loops these functions contain.

The hierarchy makes it simple to adjust migration granularity. For instance, if we have a profile with each entry corresponding to a load statement, using the hierarchy, we can easily find out the entries corresponding to all the loads in a loop or a function and migrate the profile at these larger

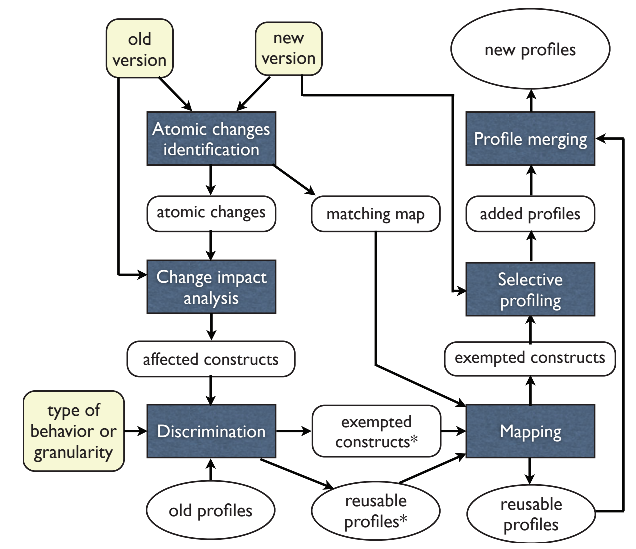

Figure 4. A profile migration framework.

granularities. It is especially useful for nesting behaviors as we will show later.

For the norm of identity to take effect in profile migration, the default profiler needs to be changed so that the entries in the output have the universal IDs of their corresponding constructs labeled. The change usually involves just simple ID replacements.

The universal ID and construct hierarchy provide a representation that unifies the multiple profile migration steps into a coherent process, simplifying the coordination among them and enabling easy adjustment of migration granularity. We next describe each of the modules in ProfMig.

4.3 Modules in ProfMig

ProfMig separates concerns into six modules, as the grey boxes in Figure 4 represent. These modules work in a modular but coherent way. Upon the norm of identity, they materialize the functions of the three stages outlined earlier in Figure 2, offering a uniform, extensible framework for migrating a variety of profiles. This section presents each of the six modules by following the structure shown in Figure 4.

4.3.1 Identifying Atomic Changes

The first component detects atomic changes between the two versions of the target program. *Atomic changes* refer to code modifications at the semantic level that is amenable to analysis [15]. Changes—such as, a local reordering of the definitions of two functions—that affect no semantics of the program are not considered.

Identifying atomic changes is a classic program analysis problem. A representative approach is to use dynamic programming to compare the syntactic parsing trees of two programs [6]. Similar techniques have been implemented in some modern tools (e.g., compare++, SmartDifferencer.) An example is the exploitation of Eclipse project management to derive atomic changes for Java programs [15]. Overall, even though identifying the precise set of atomic changes may be still difficult due to language complexities, mature tools exist for deriving a reasonable approximation set.

Besides identifying atomic changes, this module also outputs a map between the two versions. With universal IDs, the map indicates the IDs of the statements in the new version that each unchanged statement in the old version matches. This map is useful for profile reuses in the merging module, as Section 4.3.4 shows.

4.3.2 Change Impact Analysis

The second module is to find out the impact of the atomic changes identified by the first module. As a well studied topic, change impact analysis has been used for selecting test cases in software testing and guiding software development. In this work, we introduce change impact analysis into profile migration by addressing the two complexities mentioned in Section 3.3, namely the dependence of change impact on the type of profiles and program contexts, and the various applicabilities of different impact analysis techniques.

For the first complexity, because of the difficulty in addressing the two kinds of dependences at the same time, we separate the concerns to circumvent the difficulty. By leaving the influence of profile types to a separate module to consider, the design allows a profile-oblivious impact analysis module to be used. The module only tries to report all the constructs whose behaviors (no matter of what types) are potentially affected by the version update. A follow-up module (the *discrimination* module in the next subsection) translates the report to profile-specific conclusions. This design has multi-fold benefits. First, many existing impact analysis tools can be easily plugged into the system without much change required. Second, enhancement of impact analysis becomes more focused because the main factor to consider is just context. Finally, it gives the migration system good extensibility. Making the system work for a different type of profile needs no change to the impact analysis but several invocations of the interface functions of the discrimination module, which will be explained in the next subsection.

For the second complexity, from some empirical studies, we derive a simple adaptive scheme to select the change impact analysis suitable for a given migration task. In our empirical studies, we exclude dynamic impact analysis techniques as they typically incur substantial overhead. We concentrate on the following three techniques, which are representatives of static impact analysis with a spectrum of aggressiveness.

- **BA:** This refers to the classic method by Bohner and Arnold [1]. It marks all functions downstream from (i.e., invoked after) a changed function as affected. It is an intuitive design. But when the main function of a program is changed, the method will mark all functions as affected, giving the most conservative result.

- **AG:** This is an aggressive method. Like the BA method, it also works at the level of functions, but only marks functions containing atomic changes as affected.
- **SL:** This method works through static program slicing. We use CodeSurfer[3] for it. The analysis does inter-procedure data and control flow dependence analysis to identify all the statements affected by a change. Its result is always sound—that is, all statements affected by a change will be marked. It is more precise than the first two types of impact analysis. However, due to code ambiguities caused by aliases and pointers, its result is often more conservative than necessary.

As Section 6 will show, experimental results on the three techniques suggest that when versions differ substantially, SL is preferable; otherwise, AG is more desirable. We integrate this rule into ProfMig to derive a simple hybrid impact analysis.

- **Hybrid:** It uses the fraction of atomic changes over the entire program (in terms of the number of statements) as the metric of *significance of changes*. If that ratio is higher than a threshold (10% in our setting), it uses SL for impact analysis. Otherwise, it uses AG. This selection policy is simple. It is possible to design some more sophisticated policies, by for instance, giving different weights to different types of statements and different kinds of changes. In this study, we find that the simple policy works fine and use it for our experiments. But for extensibility, some interfaces are provided for reconfiguring the policy by adding weights to the types of statements and changes.

In addition, ProfMig provides some standard interface for users to add other impact analysis methods. Some simple changes may be needed to make to the method so that it works with universal IDs.

4.3.3 Discrimination

This module tries to identify the reusable entries in the old profiles and indicate the constructs that need to be reprofiled. The discrimination employs the output from the impact analysis module. A user indicates either the type of behaviors in the profile or the level of granularity the user desires for profile migration. In the former case, the discrimination module automatically infers the appropriate granularity by checking the behavior against its built-in knowledge base. The knowledge base is derived manually, containing a list of typical behaviors—currently including loop tripcounts, function calling frequencies, call graphs, load/store strides [3, 20], statement-level data reuse distances [5]—and their corresponding, finest migration granularities that are appropriate. The knowledge base can be easily extended through some interface. When the specified behavior is not in the knowledge base, the module asks the user to indicate the granularity directly. The indication is in form of an integer, equaling the level in the construct hierarchy (described in Section 4.2) that the granularity corresponds to.

The granularity decided by the knowledge base or users' indication (denoted as G_B) may differ from the granularity (denoted as G_I) used in the report from the impact analysis module due to the particular implementation of the impact analysis. The discrimination always uses the larger of them (e.g., the one at the higher level of the hierarchy described in Section 4.2.) This is because when $G_B < G_I$, no impact information exist at the level of G_B for profile migration, while when $G_B > G_I$, migration at the G_I level either violates the restrictions given in the knowledge base (and hence unsafe) or the preference of the user. When $G_B > G_I$, an impact report at the level G_B is derived from the report produced by the impact analysis module. The derivation is simple: A construct at G_B level is affected by a change if any G_I construct it contains is affected. The norm of identity and hierarchy makes the derivation possible.

The discrimination module contains an iterator, which goes through the entries in the profile construct by construct (at the granularity of $max(G_I, G_A)$), and labels an entry as reusable when the construct does not belong to the affected construct set. Those entries form the *reusable profiles**. Meanwhile, it puts the IDs of those constructs into an exempted set (the *exempted constructs** in Figure 4) so that they will be exempted from reprofiling in the later modules.

The iterator is implemented based on the construct hierarchy such that it can traverse all constructs at an arbitrary level higher than the granularity level in the profile. It builds on a base-level iterator, which can parse the output file of the profiler of interest and enumerate its entries. The base-level iterator is an interface to be instantiated by users because of the various possible formats of a profile. It also carries a boolean property "order", indicating whether the entries in the profile is order-sensitive, and a boolean property "nesting", indicating whether the entries have nesting relation. When both are true, every item enumerated by the iterator carries the scope of its corresponding construct (e.g., the function execution span in Figure 5.) If a profile has no scope labeled, the scope fields of the iterated entries are null, and the discrimination uses the entire profile of a run as the migration granularity.

4.3.4 Mapping, Selective Profiling, and Merging

The Mapping module translates the construct IDs in the *reusable profiles** and *exempted constructs** to the IDs of corresponding constructs in the new version of the program, outputting *reusable profiles* and the *exempted constructs* set. It does that by using the *matching map* produced by the module of atomic changes identification.

The follow-up module, Selective Profiling, recollects the profiles for the new version of the program without profiling the constructs in the *exempted constructs* set. It reuses the

[3] http://www.grammatech.com/products/codesurfer

main foo fi fi fi fee fj

Figure 5. A function call sequence with function execution span labeled by line segments. The labels expose the nesting relation among the call subsequences.

default profiler. Because the default profiler can be in various form and written in various languages, users' changes to the profiler is needed to add such a selective profiling feature. The change usually involves simple modifications to the program instrumentor so that it only instruments the constructs in the non-exempted construct list.

The final module, Profile Merging, combines the recollected profile data with the reusable profiles to form the profiles for the new version of the software. For ordered profiles, it needs to find the appropriate positions of the recollected data in a new profile (when the granularity is lower than the level of a program) by checking the scopes labeled in the old profiles.

4.4 Treatment to Other Complexities

In this part, we describe how ProfMig treats the other complexities.

Order and Nesting As the previous section points out, for an order-sensitive profile, the correctness of migration results can be seriously compromised if the order and nesting are not carefully handled. One solution is to treat the entire sequence in such a profile as a single unit for migration. It avoids the safety issue, but any change in the sequence will result in a recollection of the entire profile. A second option is to employ the nesting information for a finer-grained migration. Recall the example in Figure 1. A migration oblivious to the order of function call sequences may end up producing wrong call sequences. If we can have some labels in the profile marking the span of a function execution as illustrated in Figure 5, we will know the nesting relation between the subsequences “fi fi fi” and “foo”, and between “fj” and “fee”. After recollecting the order of foo and fee in the changed $main$ function, we can determine the correct positions of the two subsequences in the new profile, getting the correct profile “main fee fj foo fi fi fi”. Notice that the two subsequences “fi fi fi” and “fj” are repositioned but not recollected because their caller functions foo and fee are not affected by the code change. As Section 4.3.3 mentions, ProfMig adopts the second option, addressing the problem by associating the profile iterator in the discrimination module with two properties “order” and “nesting” and making every item enumerated by the iterator carry the scope of its corresponding construct.

Level of Abstraction Section 3.2 mentions that a profile with a higher level of abstraction causes difficulty for migration, due to the interplays among program elements incurred by the abstraction. The solution is to lower the abstraction level by changing the profilers. The change is usually simple as the profilers often collect the lower level behaviors already in order to derive the higher-level behaviors. For instance, for the hottest methods profiling, the profiler can be easily modified such that it outputs the calling frequencies of all functions and uses a separate script to identify the K most frequently invoked functions. In some cases, the change may result in too much space usage; compression may help. A general rule is that abstractions that combine the behaviors of multiple units usually introduce dependences and hence obstacles for profile migration. Such abstractions should be avoided if possible to make the profile amenable for migration.

5. Evaluation Metrics

Being a preliminarily understood topic, profile migration lacks studies in what metrics suit its evaluation. A simple examination of the accuracy of the produced profile is insufficient because it cannot reflect the detailed effects (e.g., the quality of the profile discrimination.)

In this work, we introduce a set of metrics to achieve a comprehensive assessment. For clarity, we use the following example for explanation.

A Loop Example: A profile migration system tries to migrate loop trip-count profiles of a program from version V1 to V2. An old profile F_o contains 100 entries corresponding to 100 loops in the program. The migration system identifies 80 of the entries as reusable (denoted as set U) and the other 20 as not (denoted as set $\bar{U}$.) A manual examination confirms that only 75 of the entries (denoted as set TP for true positive) in U are truly reusable, and only 10 of the entries (denoted as set TN for true negative) in $\bar{U}$ are truly not reusable. The 5 mistakes in U are *false positive* cases, and the 10 mistakes in $\bar{U}$ are *false negative* cases, denoted as set FP and FN respectively. Based on its discrimination result, the system reprofiles the loops in $\bar{U}$ and merges with the old data of the loops in U to produce a merged profile for V2, denoted as F_m. Let F_c represent the correct profile of V2.

We find five metrics useful, one for the accuracy of the merged profiles, three for profile discrimination, and one for time saved by profile migration.

(1) **Acc.** This metric is for accuracy, computed as the ratio between the number of the entries in the generated profile that are correct and the total number of entries in the real profile. Here, each entry refers to one unit in the profiles. For the loop example, an entry is the trip-count of a loop. The Acc of the above loop example is (75+20)/100=95% because only loops in TP and $\bar{U}$ have correct trip-counts in F_m. (Loops in $\bar{U}$ are reprofiled.)

(2) **Precision, Recall, F-measure.** These three metrics are borrowed from information retrieval. They measure the quality of the profile discrimination. *Precision* is defined as $|TP|/|U|$, showing the fraction of the claimed reusable entries that are truly reusable. *Recall* is defined as

$|TP|/(|TP|+|FN|)$, showing the fraction of the reusable entries that are successfully identified. *F-measure* combines precision and recall together to give an overall measure of the effectiveness of the discrimination, computed as (2·precision·recall)/(precision+recall). For all these metrics, the higher the better; the upper bounds are all 1.

(3) **TS.** This metric is the fraction of the time saved by the profile migration. It equals the total reduction of profiling time of all runs (with the profile migration overhead counted) divided by the total profiling time of the new version when migration is not used.

6. Experiments

In this section, we examine the effectiveness of the techniques in helping migrate two types of profiles. The first is loop trip-counts, each entry of which reports the number of total iterations of a loop in an execution. The second is dynamic call graphs, each entry of which reports the caller of a function and the number of times that caller calls that function. Even though both profiles are important for feedback-driven program optimizations, they differ significantly in granularity and structure: The former is in a linear structure with loops organized in a list, and has loop as its finest migration granularity, while the latter is in a graphic structure with strong connections between callers and callees, and has function as its finest granularity. Moreover, their profilers are implemented on two different infrastructures: LLVM [12] for the former, and Gprof for the latter. These differences help test the flexibility of the migration system.

6.1 Methodology

When finding benchmarks, we focus on the real-world software that is the source for SPEC CPU benchmarks. They cover a variety of domains. Among all the programs we have examined, five are still actively maintained so that we can find two versions of each of them that run successfully on our platform (Suse Linux 2.6.37.6, Intel Xeon X5570, 2.93GHz.) Table 1 lists the characteristics of the programs. All these programs are written in C/C++. They range from utility programs to quantum computing, circuit routing, and games.

Because typically software version update goes through an incremental process, profile migration *usually applies to two close-by versions*. However, even between the two versions differing only in minor version numbers, some of the programs (*vpr* and *crafty*) still show large differences as shown by the two rightmost columns of Table 1. The similarity in the table is defined as the fraction of the entries in the actual profiles of the higher version that also appear in the profiles of the lower version. We have experimented with five difference inputs for each of the programs. The similarities of these runs may differ; the table shows the average.

For the time measurement, we count in all migration overhead. We conduct five repetitive runs to get the average as the final reported number.

6.2 Results

Overview Overall, the experiments show that for most of the programs, ProfMig can save a substantial fraction of profiling time with high profile accuracy preserved. For loop trip-counts, it cuts profiling time by more than 90% for three programs, 0–15% for the other two programs. The migration results have accuracies all higher than 88%. More than 90% of the profile entries that ProfMig marks as reusable are indeed reusable. The amount of time savings in call graphs migration is relative less (48% versus 61% on average) than in the loop case because of the strong connections between callers and callees in the profiles. But the overall results still consistently show the promise of ProfMig in enabling effective migrations of different types of profiles.

The results also indicate some opportunities for future improvement. The most prominent is that ProfMig misses some, sometimes a large number of, reusable profile entries. The main reason resides in the conservativeness of the static impact analysis. The consequence is most obvious for programs that contains lots of aliases and pointers and a good volume of code changes from their old versions (exemplified by the benchmark *vpr*.) It suggests the need for better impact analysis techniques to be developed in the future.

Detailed Analysis Table 2 reports the detailed experimental results. It shows the comparison when ProfMig uses four kinds of impact analysis (listed in Section 4.3.2.) The metrics used in the table are described in Section 5.

For loop trip-counts, the migration based on BA gives almost perfect accuracy for most programs; confirmed by the high precision rates. An exception is *libquantum*. Libquantum is a program containing some indeterminism for the use of random numbers. The randomness causes some profile entries to differ even though their corresponding code is not affected by the version update. The BA-based migration saves some profiling time on *libquantum* but not on other programs. The zero recall rates on the other four programs reflect the reason for no time savings on them: The main function in all these four programs are changed in the version update. Although most of the changes are not critical for many loops' trip-counts, the BA-based impact analysis conservatively mark all the functions of those programs as affected by the changes. No reusable profile entries are left. The F-measure for all the four programs is 0. The code changes in *libquantum* happens in some functions that are at the lower levels of the call chain, and is hence more amenable for BA-based profile migration. For the 0.9 recall rate on *libquantum*, the migration saves over 99% profiling time because the reused loops have higher trip-counts and hence would require more reprofiling time than the other

Table 1. Characteristics of Benchmarks

Software	Description	Versions	Numbers* of			Changed	Similarity	
			code lines	loops	functions	functions	loops	call graphs
bzip2	data compressor	1.01; 1.03	7388; 7662	260; 253	123; 133	31	98.73%	100%
crafty	computer chess program	23.1; 23.2	34907; 35056	542; 538	181; 182	19	78.40%	43.54%
libquantum	computer quantum simulator	1.0; 1.1	4883; 4938	134; 135	144; 146	5	88.46%	81.33%
parser	english parsing tool	4.0; 4.1b	18609; 18923	765; 765	642; 642	3	93.68%	90.77%
vpr	circuit placement and routing	4.22; 4.30	22990; 25399	575; 645	419; 430	21	61.35%	54.87%

*: The numbers are of the two versions of the software with the number of the higher version following that of the lower version.

loops. Overall, BA-based migration offers high accuracy but low time savings.

The migration based on SL gives accuracies comparable to those from BA-based migrations. It saves much more profiling time than the BA-based migration on *crafty*. However, its recall rates still remain below 30% for most programs. Manual analysis confirms the reason: The slicing method, for producing sound results, reports much larger impact scopes than the actual for an atomic change for the aliases and pointers in the program. Given the considerable number of changes between two versions as shown in Table 1, the method results in a large number of false negatives (i.e., an unaffected construct is labeled as affected.) Its F-measures vary from 0.06 to 0.85.

The migration based on AG gives the highest time savings for its aggressiveness in profile reuses. It recalls over 95% of the entries that are truly reusable on programs *libquantum*, *parser*, and *vpr*, 78% for *crafty*, and 35% for *bzip2*. The relatively lower recall rate on *bzip2* is because many code changes in that program do not affect the loop trip-counts, but AG-impact analysis conservatively assume the profiles of all loops in a changed function are not reusable. The time savings by this method ranges from 72% to nearly 100%, with an average 93%. It is not as precise as the other two methods, with precision ranging from 0.7 to 1. The accuracy of the produced profiles is above 88% for three programs. For the two programs with lower cross-version similarities, *crafty* and *vpr*, the accuracy is 79% and 67%.

The bottom section of Table 2 reports the results of the migration based on the hybrid impact analysis. By applying a suitable impact analysis to each program, the approach gains the best of both worlds, producing accurate profiles with accuracy from 89% to 100% with an average 97%. It meanwhile brings large time savings, with an average 61%.

The results on the call graphs show similar patterns as the loop trip-counts do with slightly smaller time savings. Its larger granularity and the coupling relations between callers and callees result in relatively a lower fraction of reusable profile entries. It is worth noting that no changes are made to ProfMig to support the two different types of profiles; it adapts to the profiles automatically.

The aggressive method is not as sound as the slicing method, and may produce some false positives. In software testing, which is the traditional chief usage of impact analysis, a false positive may prevent some affected construct from being retested and hence cause threat to the software reliability. But in the context of profile migration that mainly assists program optimizations, the consequence is much less serious. It may result in some errors in the produced profiles, which may then mislead the optimizer somehow and ultimately affect the quality of the produced code, but not the reliability or correctness of the software. Moreover, many optimizations tend to be resilient to a certain degree of profile errors. So generally, false positives in impact analysis are less harmful for optimization-oriented profile migration than for software testing. We note that unlike the local effects by the false positives in impact analysis, the pitfalls mentioned in previous sections of this paper are at the fundamental level. Incorrect treatment may result in whole-profile errors. For instance, if order is not observed in the migration of function call sequences, the produced profile may be entirely wrong.

Overall, the experiments demonstrate that profile migration is feasible in practice. The migration framework proposed in this paper is promising to work for different kinds of profiles. Assisted with adaptive impact analysis, the framework can produce accurate profiles with large time savings.

7. Discussions

Profile migration may produce profiles with some errors. Our experience on feedback-driven optimizations shows that the real usage of a profile often has a certain degree of error tolerance: Some amount of errors in the profile do not affect optimization results much.

In the comparisons, we have focused on measuring the direct quality of the profiles themselves. We did not use the usage of the profiles for comparison mainly because that metric largely depends on how sensitive the particular usage is to profile accuracy. Since the sensitivity differs on different usages, the errors in a generated profile may show a minor effect on one usage but a large effect on another usage. With that said, adding the results on some usages may still help to show the ultimate influence of the profile errors. It is part of our future work.

In this study, we have used sequential programs for evaluation. The profile migration techniques and framework can be applied to parallel programs as well. A necessary extension is that the impact analyzer should be equipped with the

Table 2. Profile Migration Performance Results

Impact analysis	Software	Loop trip-counts					Call graphs				
		Acc	TS	Precision	Recall	F-measure	Acc	TS	Precision	Recall	F-measure
BA*	bzip2	100%	0%	1.0	0.0	0.0	100%	0%	1.0	0.0	0.0
	crafty	100%	0%	1.0	0.0	0.0	100%	0%	1.0	0.0	0.0
	libquantum	88.8%	99.1%	0.9	0.9	0.9	81.1%	64.2%	0.8	1.0	0.9
	parser	100%	0%	1.0	0.0	0.0	100%	0%	1.0	0.0	0.0
	vpr	100%	0%	1.0	0.0	0.0	100%	0%	1.0	0.0	0.0
	Average	97.8%	19.8%	1.0	0.2	0.2	96.9%	12.8%	1.0	0.2	0.2
SL*	bzip2	100%	0%	1.0	0.04	0.08	100%	0.0	1.0	0.0	0.0
	crafty	100%	15.1%	1.0	0.3	0.4	100%	0.0	1.0	0.1	0.2
	libquantum	89.1%	100%	0.9	0.8	0.9	81.1%	64.2%	0.9	1.0	0.9
	parser	98.1%	0%	0.9	0.2	0.3	100%	0.0	1.0	0.02	0.04
	vpr	100%	0%	1.0	0.03	0.06	100%	0.0	1.0	0.0	0.0
	Average	97.4%	23.0%	1.0	0.3	0.4	96.2%	12.8%	1.0	0.2	0.2
AG*	bzip2	99.8%	91.2%	1.0	0.4	0.5	100%	91.3%	1.0	0.4	0.5
	crafty	79.4%	72.5%	0.7	0.8	0.8	64.7%	0.1%	0.5	0.6	0.5
	libquantum	88.8%	100%	0.9	1.0	0.9	81.1%	64.2%	0.8	1.0	0.9
	parser	93.7%	100%	0.9	1.0	1.0	95.4%	85.8%	1.0	0.9	1.0
	vpr	67.4%	100%	0.7	1.0	0.8	61.5%	0.001%	0.5	0.8	0.7
	Average	85.8%	92.7%	0.9	0.8	0.8	80.5%	42.3%	0.8	0.7	0.7
Hybrid	bzip2	99.8%	91.2%	1.0	0.4	0.5	100%	91.3%	1.0	0.4	0.5
	crafty	100%	15.1%	1.0	0.3	0.4	100%	0.0	1.0	0.1	0.2
	libquantum	88.8%	100%	0.9	1.0	0.9	81.1%	64.2%	0.8	1.0	0.9
	parser	93.7%	100%	0.9	1.0	1.0	95.4%	85.8%	1.0	0.9	1.0
	vpr	100%	0%	1.0	0.03	0.06	100%	0.0	1.0	0.0	0.0
	Average	96.5%	61.3%	1.0	0.5	0.6	95.3%	48.3%	1.0	0.5	0.5

∗ BA: a classic method by Bohner and Arnold [1]. ∗ SL: forward slicing by Codesurfer. ∗ AG: aggressive function-level analysis.

semantics of some parallel constructs (e.g., locks). A detailed study is our future work.

8. Related Work

There are several studies closely related with this work. Wang and others present a binary matching tool, named BMAT, which tries to reuse stale profiles through binary matching [19]. It differs from our study in three main aspects. First, a main feature of our study is the general, modular design—including the norm of identity and the hierarchy and the whole system design—which is essential for handling various profiles with different impact change analysis techniques used. The BMAT work only focuses on a specific type of profile. Second, the BMAT work relies on binary basic block matching rather than change impact analysis. The profile of a code region that is not changed is regarded as reusable in BMAT, but not necessarily so in our method, depending on whether it is affected by the code changes to other regions. Third, BMAT works on binary code while our method works on source code. The different levels entail different complexities (e.g., the needs for the norm of identity and the hierarchy to allow flexible granularity adjustment in our work; BMAT has a fixed granularity, basic blocks.)

Zhang and Gupta have studied the matching of the execution histories of program versions [21]. They design some matching algorithms that match the entries in the histories by examining these histories (rather than code) directly. It does not generate profiles, nor does it do impact change analysis. The Symdiff work by Lahiri and others offers a language-agnostic tool for identify behavioral differences between two versions of a program [10]. It can serve as a component in our framework by, for example, replacing the Chianti for atomic change analysis. Being language-agnostic, it may help expand the applicability of our framework.

Some techniques used in our profile migration are derived from some other domains. Atomic change identification has been studied since 1980s. An example is the work by Horwitz in 1989, which uses dynamic programming to compare the syntactic parsing trees of two programs [6]. The goal of the analysis is to identify the code changes between two versions that are relevant to the semantic of the program. An unchanged construct, even though its execution may become different due to the impact of the changes in some other constructs, is not identified in such analysis.

Change impact analysis extends atomic change identification to find all constructs whose executions are potentially affected by code changes in the program. Previous techniques fall into three categories. The first relies completely on static information. Examples include the usage of static call graph described by Bohner and Arnold [1] and the exploitation of various sorts of relationships between classes in an object relation diagram by Kung and others [9]. The second category employs only dynamic information. Examples include the usage of whole-path profiling [11] in the

PathImpact by Law and Rothermel [13]. The third category combines both types of information. Examples include the *CoverageImpact* by Orso and others [14], and the *Chianti* by Ren and others [15]. Change impact analysis is an important component in profile migration. However, by itself, it is not sufficient for identifying the reusable entries in a profile. As we have shown, even if a construct's execution is not affected by code changes, its entry in the profile may not be directly reusable (reprofiling or repositioning would be needed as the call sequence example in Section 3 shows.) The reusability depends on the order, nesting, and other properties of the profiled behaviors that have been discussed in this paper.

An orthogonal way to reduce profiling cost is through sampling. Examples include the sampling techniques for program optimizations (e.g. [4, 7, 16]) and debugging (e.g., [2, 8]). Sampling trades accuracy for efficiency. It is complementary to profile migration. They can be used synergistically to achieve a better accuracy and efficiency.

9. Conclusion

In this paper, we have described a systematic study on program behavior profile migration across software versions. We explore the various factors related with profile migration, revealing the effects imposed on profile reusability and migration granularity by the properties of the profiled behaviors, profile formats, and impact analysis. The exploration leads to some fundamental understanding to the profile migration problem, yields some general guidelines for profile migration, and points out some potential pitfalls. We propose a six-module framework for profile migration. It uses a norm of identity and hierarchy for representation to unify the multiple profile migration steps into a coherent process. It isolates concerns and allows easy customization and extension. Its applications to loop trip-count profiling and dynamic call graph profiling demonstrate the feasibility and promising potential of profile migration. This work is expected to open many new opportunities for removing the barriers for the application of profiling in software development and optimizations.

Acknowledgment

We thank Ben Zorn for his suggestions to this work and great help in enhancing the final version of this paper. We owe the anonymous reviewers our gratitude for their helpful comments to 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] S. A. Bohner and R. S. Arnold. An introduction to software change impact analysis. In *Software Change Impact Analysis*, pages 1–26. IEEE Computer Society Press, 1996.

[2] M. D. Bond, K. E. Coons, and K. S. McKinley. Pacer: Proportional detection of data races. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[3] B. Calder, P. Feller, and A. Eustance. Value profiling. In *Proceedings of International Symposium on Microarchitecture*, 1997.

[4] W. Chen, S. Bhansali, T. M. Chilimbi, X. Gao, and W. Chuang. Profile-guided proactive garbage collection for locality optimization. In *Proceedings of PLDI*, pages 332–340, 2006.

[5] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, CA, June 2003.

[6] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. Technical Report 895, University of Wisconsin-Madison, 1989.

[7] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[8] G. Jin, A. V. Thakur, B. Liblit, and S. Lu. Instrumentation and sampling strategies for cooperative concurrency bug isolation. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2010.

[9] D. C. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen. Change impact identification in object oriented software maintenance. In *Proc. of the International Conf. on Software Maintenance*, 1994.

[10] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebelo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Proc. of the Computer Aided Verification (CAV '12)*, 2012.

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

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

[13] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *Proc. of the International Conf. on Software Engineering*, 2003.

[14] A. Orso, T. Apiwattanapong, and J. Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. of European Software Engineering Conf. and ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE03)*, 2003.

[15] X. Ren, S. Fenil, T. Frank, R. G. Ryder, and C. Ophelia. Chianti: A tool for change impact analysis of java programs. In *the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[16] D. Schuff, M. Kulkarni, and V. Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, pages 53–64, 2010.

[17] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *Proceedings of the ACM SIGPLAN Conference on Principles of Programming Languages (POPL)*, pages 55–62, 2007.

[18] F. Tip. A survey of program slicing techniques. *J. of Programming Languages*, 3(3), 1995.

[19] Z. Wang, K. Piece, and S. McFarling. BMAT – a binary matching tool for stale profile propagation. *Journal of Instruction-level Parallelism*, 1(6), 2000.

[20] Y. Wu. Efficient discovery of regular stride patterns in irregular programs. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[21] X. Zhang and R. Gupta. Matching execution histories of program versions. In *Proc. of European Software Engineering Conf. and ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE05)*, 2005.