Progressive Profiling: A Methodology Based on Profile Propagation and Selective Profile Collection

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters.

Citation

Citable link
http://nrs.harvard.edu/urn-3:HUL.InstRes:25104736

Terms of Use
This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRes:dash.current.terms-of-use#LAA
Progressive Profiling: A Methodology Based on Profile Propagation and Selective Profile Collection

Zheng Wang

TR-11-01
November 2001

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

© 2001 - Zheng Wang
All rights reserved.
Abstract

In recent years, Profile-Based Optimization (PBO) has become a key technique in program optimization. In PBO, the optimizer uses information gathered during previous program executions to guide the optimization process. Even though PBO has been implemented in many research systems and some software companies, there has been little research on how to make PBO effective in practice.

In today’s software industry, one major hurdle in applying PBO is the conflict between the need for high-quality profiles and the lack of time for long profiling runs. For PBO to be effective, the profile needs to be representative of how the users or a particular user runs the program. For many modern applications that are large and interactive, it takes a significant amount of time to collect high-quality profiles. This problem will only become more prominent as application programs grow more complex. A lengthy profiling process is especially impractical in software production environments, where programs are modified and rebuilt almost daily. Without enough time for extensive profiling runs, the benefit from applying PBO is severely limited. This in turn hampers the interest in running PBO and increases the dependency on hand tuning in software development and testing.

In order to obtain high-quality profiles in a software production environment without lengthening the daily build cycle, we seek to change the current practice where a new profile must be generated from scratch for each new program version. Most of today’s profiles are generated for a specific program version and become obsolete once the program changes. We propose **progressive profiling**, a new profiling methodology that propagates a profile across program changes and re-uses it on the new version. We use static analysis to generate a mapping between two versions of a binary program, then use the mapping to convert an existing profile for the old version so that it applies to the new version. When necessary, additional profile information is collected for part of the new version to augment the propagated profile. Since the additional profile collection is selective, we avoid the high expense of re-generating the entire profile. With progressive profiling, we can collect profile information from different generations of a program and build a high-quality profile through accumulation over time, despite frequent revisions in a software production environment.

We present two different algorithms for matching binary programs for the purpose of profile propagation, and use common application programs to evaluate their effectiveness. We use a set of quantitative metrics to compare propagated profiles with profiles collected directly on the new versions. Our results show that for program builds that are weeks or even months apart, profile propagation can produce profiles that closely resemble directly collected profiles. To understand the potential for time saving, we implement a prototype system for progressive profiling and investigate a number of different system models. We use a case study to demonstrate that by performing progressive profiling over multiple generations of a program, we can save a significant amount of profiling time while sacrificing little profile quality.
Chapter 1. Introduction

Computers, as well as the way that people use them, have evolved continually in the past decades. In the 1960s and 1970s, computers were mostly confined within the science and engineering community. Most machines ran variations of the UNIX operating system and were used predominantly for scientific computations. Programs usually ran in batch style, with no human interaction during execution. Even though compiler optimizations were designed to improve application performance, there was often an implicit expectation that programmers would write programs that were efficient. This is evident from the fact that programmers often wrote the “core computation” portion of their programs in assembly code for maximum efficiency.

In the 1980s and 1990s we experienced the tremendous popularization of personal computers and Windows-style operating systems. These computer systems are frequently used for interactive tasks instead of batch-style scientific computation. Large, complex and GUI-based applications have become ubiquitous, and are growing even larger and more complex due to advances in hardware technology. These developments have had significant impact on the software industry in terms of program optimization.

First of all, the extremely competitive software market and the rapid development in hardware technology often force software companies to speed up the process of developing a new generation of applications. This poses a conflict of interest with the general increase in application complexity. In order to get a software product on the market as soon as possible, the efficiency of application programs often takes lower priority to implementing marketable features and fixing bugs. With limited timeframe, the expectation for program efficiency is largely placed on compiler optimizations.

Secondly, to design and implement a complex application takes a large team of programmers. The coordination between programmers is often considered more important than the efficiency of the program code. Object-oriented programming languages such as C++ and Java are frequently used to develop large applications because the modularization of program organization and data structure greatly facilitates collaborative programming. However, the flip side of producing correct codes easier and faster is that we have to rely on powerful compilation and optimization tools to produce efficient binary code.

Furthermore, new hardware features such as superscalar and superpipelining CPUs, deeper memory hierarchies and branch prediction hardware provide a plethora of new opportunities for compiler optimization. A general increase in the average program size also brings code placement optimizations [Pettis and Hansen 1990] to the forefront. These new factors all add to the dependency on optimization for good performance.

In the past decade or so, the development in computer system design and compiler research has led to the idea of Profile-Based Optimization (PBO). Traditionally, compiler optimizations are based on information gathered from static analysis of program code. In other words, the optimizations are completely blind to how the program is used in the past or the future. PBO, in contrary, is a class of optimizations driven by an understanding of the program’s runtime behavior.

Profiling is the process of gathering runtime information that is later used by an optimizer to understand a program’s prevalent behavior. The information that describes the program’s
behavior is usually stored in a file known as a *profile*. Depending on the specific optimization, a profile can contain a variety of information, such as basic block execution counts, branch history, cache miss statistics, etc. Profile information allows the optimizer to customize the program code according to the way it is used. It also introduces new optimization opportunities, especially on procedure level and basic block level, that are not possible using only static analysis of the program. The benefits of PBO are dependent upon having adequate profiles that accurately describe the program behavior. For PBO to be most effective, the profile needs to summarize how a large portion of the program is most frequently used.

This thesis focuses on the issue of generating high-quality profiles in software development environments where application programs are frequently modified and rebuilt. In today’s practice, profiles typically become obsolete once the binary program changes. As a result, the amount of profile information and therefore the benefit of optimization are severely limited by frequent revisions of the program. We present a new methodology called *progressive profiling*, which extends the life of profile information beyond the time of program revisions. This enables software developers to receive more benefits from PBO by accumulating high-quality profiles over time despite frequent program changes. In the rest of this chapter, we first analyze the shortcomings of current profiling practice, then outline the high-level design of progressive profiling.

### 1.1. Limitations in Current Profiling Methodology

Researchers have developed several different methods of profile collection. Instrumentation tools such as ATOM [Srivastava and Eustace 1994] rewrite the program executable and insert code that collects desired information during program execution. Instrumentation can be used to collect many types of information accurately, but can cause significant slowdown in program execution and distortions in program behavior. Some research systems such as DCPI [Anderson et al. 1997] and Morph [Zhang et al. 1997] utilize system interrupts to collect program execution information without modifying the program code. This method of statistical sampling is non-intrusive and has very low overhead, but does not monitor the entire program execution.

Despite all the research on profiling mechanisms, today’s PBO research largely ignores the question of how to obtain profiles that are adequate for the commercial use of PBO. In most research systems, there is little restriction on the profile collection process. Intrusive methods such as instrumentation can be used without worrying about user tolerance, and the profiling process can go on for a long time if deemed necessary. This methodology is often not practical in commercial situations. Figure 1.1 (*see next page*) demonstrates the typical PBO methodology in a research environment.

In industry practice, profile collection can become a difficult issue. In most of today’s PBO systems, profile information is directly associated with the binary program instead of an intermediate representation or the source code. The profiles become obsolete once the binary program changes, be it a result of source code modification, re-compilation with different settings, or binary optimization. Consequently, the time for profile collection is limited by the time span between two program revisions. With each new version of the program, profile collection has to be repeated from scratch to re-generate the profiles we need.
In a typical research system, the program source code doesn’t change. Profiles are gathered by either running pre-selected workloads or monitoring real users. There is virtually no time restriction for profile collection. Once the optimization is performed, the profiling process generally starts over.

In a typical software production system that employs PBO, the profiling and optimization is included as part of the nightly building process. There is little time available to generate profiles for each build, on the orders of hours or even minutes. Due to time restraint, profiles are usually generated using script-driven automatic tests. Profiles are no longer useful once a new build is generated. With little time devoted to profile collection, the benefit of PBO is severely limited by the quantity and quality of profiles.
In a software production environment, programs are modified frequently, usually with a new version generated every day. Once a new build is compiled and optimized, there is a pressing need to turn the program to the testing team and back to the programmers. In a typical setting, the source code is compiled overnight and new binaries are generated in early morning hours. Profiles are then collected using automated test scripts, and optimizations are performed to generate the final program build. This optimization step is important as it reduces the pressure on testers and programmers to tune the program manually. In order to make the optimized program available before the work day starts, the time available for profile collection may be limited to a matter of minutes. Figure 1.2 (see previous page) illustrates this situation.

The problem of profiling time is amplified by the fact that many of today’s interactive applications are large and complex, with an abundance of functionality and features. Therefore, there are a large number of possible ways to execute the program. Even when the same task is performed, the program execution path may be affected by factors such as location and size of the application window, position of the mouse, timing variations in user input, and system activities from other applications. For PBO to be effective, the training workload must cover a significant portion of this vast space of program behavior, while still conveying which behaviors are more likely than the others. As a result, we may need a significant amount of execution time to collect a sufficient amount of profile information.

PBO can also be conducted on an end user’s machine, although user-specific PBO is still in research stage and has not been put to practical use. The methodology of user-specific PBO is similar to the methodology in a research system in that there is no immediate time limit on profile collection. However, profile collection time can also be a matter of concern in these systems, as the cycle of profiling and optimization leads to a dilemma in terms of optimization frequency. There are two reasons for performing optimizations frequently. In order to take advantage of performance improvement from the optimization, it is desirable to optimize the program as soon as there is enough profile information to indicate how this user tends to run the program, or what we call the usage pattern of this user. Also, the system should respond by re-optimizing the program when profiles indicate changes in the usage pattern. On the other hand, since each optimization changes the binary and makes all existing profiles obsolete, frequent optimizations will limit the amount of profile information and the time span it represents. This conflicts with the goal of following the changes in usage pattern over time. Furthermore, profiling on a user’s machine is more likely to employ statistical sampling rather than instrumentation, which further limits the amount of information that can be gathered during a certain time.

In summary, many problems in today’s profile collection are directly related to the fact that profiles become obsolete once the binary program changes. This limitation poses a major challenge to putting PBO into practice, especially in software production environments. In the next section we elaborate on the importance of profiling and the consequences of inadequate profiling.

1.2. The Assumptions and Pitfalls in Profiling

Profile-based optimizers differ from traditional optimizers in that they utilize profiles. Some of these optimizers must rely on profile information to proceed, while others can run with or without profile information. In either case, the optimizer analyzes program execution record
stored in profiles to help make optimization decisions. With an effective optimizer, these decisions are expected to lead to optimal performance when the program is used the same way it was when the profile was collected. The real goal, on the other hand, is to improve the program performance in future uses. The assumption here is that the profile serves as a good prediction for the program behavior in the future. If the profile is not representative of how the program is used, PBO may not provide significant performance improvement, and may even cause performance degradation.

In reality, it is unlikely that an interactive program is used the same way every time. Indeed, many modern applications may follow a different execution path every time they are executed, due to variations in user input and time-based operations. Frequently, the program input for a program run after the optimization (referred to as the testing workload) is different from the program input for the program run(s) from which the profile is generated (referred to as the training workload). A correct optimizer choice for training workload may not be the correct choice for testing workload. Thus, the training workload should be representative of the different ways a program might be used, so that the profile is a good prediction for program behavior under the testing workload. Figure 1.3 (see next page) illustrates the issue of training and testing workloads.

In the case of a real user, the testing workload is the tasks he or she performs on the program after it is optimized, while the training workload may come from various sources depending on when and where the profiling and optimization are performed. Different models for applying PBO make various assumptions on how different users run the same program. In the traditional optimization model, an application is optimized before it is released to users. PBO may be applied using profiles collected from either automated test scripts or manual test runs. The underlying assumption is that all users use the same program in largely similar ways, and therefore the test scripts or test users can represent the potential usage of the program. In recent years, research systems such as Morph [Zhang et al. 1997] and FX!32 [Hookway and Herdeg 1997] have been designed to extend the optimization process beyond software shipment. In these systems, profiling and optimization are performed on an end user’s machine where the program is used, and can be implemented as an ongoing process. This model assumes that different users run the same program in significantly different ways, while each user maintains a relatively consistent usage pattern. In a third model that targets shared applications at corporations and schools, one copy of the program is stored, profiled and optimized on a central server. The assumption is that while different users have different ways of using a program, a related group of users have common usage patterns among them.

In a previous project by Wang and Rubin [1998], we used statistical analysis to investigate the different assumptions on user-specific profiles. We tested a collection of real-life user profiles for some common Windows NT application programs. Our results showed that users are different from each other in terms of procedures they use, and in some cases such differences have notable impact on the performance benefit from optimization. However, all tested profiles provided better optimization performance compared to using no profile or a minimal profile, suggesting that pre-shipment PBO can provide meaningful benefit. We did not have enough data to test the assumption on group-specific profiling. Overall, our results indicated that for good performance of PBO it is important to have a sufficient amount of profile information, and that the quality of profile can have notable impact on the optimization performance.
The relationship between training/testing workloads and optimization performance is complex, and can only be discussed in the context of individual optimizers and specific applications. This thesis does not aim to resolve this relationship.

1.3. Progressive Profiling

To address the problem of profiling time in software production environments, we propose a new model where profile information is preserved across program changes in order to reduce required profiling time. The centerpiece of this model is a tool that takes an existing profile associated with an old version of a binary program and converts it so that it can be used to describe expected behaviors of a new version. This process, which we call profile propagation,

---

1. We use the word “version” in a general sense. Any operation that changes the binary program produces a new version. This operation could be one of the followings: modifying the program source code and re-compiling it; re-compiling the same source code with a different compiler or different compiler settings; optimizing the binary program; other binary transformations such as program patching. In fact, profile propagation can still apply if the two “versions” are of two different programs. Our technique is general and can be applied regardless of the reason for generating a new version.
extends the lifespan of profile information and reduces the need for extensive profiling runs every time a new version is built.

It is important to note that the success of profile propagation is contingent upon the similarity between two program versions. Changes in program code may make it difficult or impossible to predict how the new version will behave. If the old version and the new version are very similar, profile propagation can likely produce a high-quality profile for the new version and may eliminate the need for any new profile collection. If some sections of the program are substantially different in the two versions, profile propagation may not yield satisfactory results for those sections. If new codes are written for the new version, we may not be able to obtain any profile information for them through profile propagation. For the latter two cases, we can use a reduced profiling process to generate additional information that augments and completes the profile. This process is called **selective profile collection**. Even if the propagated profile already describes the new version fairly well, we can still choose to perform additional profile collection and combine the newly collected profile with the propagated profile.

By repeating profile propagation and selective profile collection over multiple generations of a program, we can make the task of generating high-quality profiles a continuous process that extends beyond the life of each version. We call this methodology **progressive profiling**. Figure 1.4 illustrates the process.

![Figure 1.4. Progressive profiling.](image)

**Figure 1.4. Progressive profiling.** In this model, profiles are preserved across program revisions, which could be caused by source code modification, re-compilation or program optimization. Optionally, additional profiling runs can be administrated to gather information that augments the propagated profile. With progressive profiling, profile information can live through many generations of the program, and we can obtain a high-quality profile through accumulation over time.

---

A new term in the realm of computer science, **progressive profiling** is an existing concept in customer management and other human-related research. The term is used to describe a technique where, instead of asking a human subject many questions all at once, the researcher asks a few questions initially and then comes back later to gather additional information. By allowing the subject’s profile to grow over time, the profiling process becomes “much less intrusive” and the profiles “tend to be far more up-to-date and accurate than one accumulated in a single sitting” [Frey and Harvey 1999]. In the case of program profiling, the implications of an extended profiling process are much more complex. However, the ideas of accumulating information over time and reducing resource requirement for each step are similar.
In order to propagate profile information between two versions of a program, we use static analysis to generate a code mapping between the two binaries. This process is called **binary matching**. The code mapping is then used to guide the profile propagation. In the context of progressive profiling, binary matching can also provide information on which sections of the program have changed between the two versions. This information is needed to drive the selective profile collection. In this thesis, we present two different algorithms for binary matching, and use them to conduct experiments on profile propagation.

The process of progressive profiling involves a trade-off between profile propagation and selective profile collection. The more profile information we propagate from the old version, the less we have to collect on the new version. On the other hand, due to program changes, the propagated profile may not always accurately describe the behavior of the new version. It is useful, therefore, for the binary matching algorithm to inform us of not only sections of the program that cannot be matched, but also sections that may have changed in terms of runtime behavior. During selective profile collection, we will have the option to re-collect profile information for program sections where a change in runtime behavior is likely and profile quality confidence is low.

Progressive profiling introduces new possibilities to the process of applying PBO. Instead of being limited by the amount of time between program revisions, PBO can now be expanded to an evolving process that spans multiple generations of a program. As a result, there can be parallelism in the process of profiling and optimization. In a software production environment, frequent program revisions no longer restrict the profiling time available. Software testers can perform extensive profiling on the latest version of a program while the next version is being developed. The resulting profile can be propagated onto the new version after it is built the following morning. This is a vast improvement over running a small number of script-driven tests during an overnight build process. By propagating profiles through multiple generations, we can increase the time available for profile collection from minutes or hours to days or weeks. Similarly, in a user-specific PBO system, progressive profiling means that the need for extensive profiling no longer prohibits frequent program optimizations.

Progressive profiling also highlights the question of what a profile represents and how specific it is. In current PBO systems, a profile typically reflects program behavior during one previous run or maybe a few. In progressive profiling, a profile may include execution records from many runs of multiple program versions. Therefore, the profiles become less specific than traditional profiles, but may cover a wider range of possible workloads. Whether this is better or worse for guiding optimizations is a subject for further study and outside the scope of this thesis. The optimal balance between specific and broad information may depend on the sensitivity of the optimizer and the complexity of the program. This balance also introduces many new questions in terms of profile management. For example, how should we combine a propagated profile with a directly collected profile? Should we eliminate propagated information from many generations ago, either by sudden elimination or through an “aging” process? These questions are beyond the scope of this thesis, although we will touch on them briefly in Chapter 6 when we discuss different models of progressive profiling.

---

3 Our system is designed not to rely on the source code or additional information from the programmers or the compiler. Section 2.2 will discuss this in more detail.
1.4. Related Work on Profile Re-use

Wall [1990] explored using static analysis to estimate profiles for binary programs. Estimated profiles were found to be insufficient to replace dynamically collected profiles, except in limited cases. Meanwhile, profile estimation can potentially be used as part of the progressive profiling model. For program sections that have changed too much for profile propagation, we can use static estimation to generate their profile information if the situation prohibits us from doing additional profiling collection.

Most existing PBO systems do not address the issue of obsolete profiles. In these systems, the profiling process starts over after any program transformation. Several research systems make efforts to preserve profile information across program changes, each with different limitations. Spike [Cohn et al. 1997], an optimization system for Alpha executables, conducts a limited form of binary matching between program versions in order to re-use profile data. In Spike, profiles collected for a procedure continue to be used for successive builds if the procedure remains the same. However, Spike discards profile information for any procedure whose control flow graph has changed. In our profile propagation we preserve profile information for as much of the program as possible, even for procedures that undergo significant changes.

Some other systems use special information from the compiler or optimizer to deal with obsolete profiles. Morph [Zhang et al. 1997], a research system for automatic and continuous profiling and optimization, performs limited profile re-use in the context of program re-optimization. In Morph, profiles for a program module are generated for its special intermediate representation, which is saved in the system. When a module is re-optimized, the optimizer automatically produces a basic block mapping between the old and new intermediate representations so that existing profiles can be converted for use on the new version. There are two limitations to this approach. First is that the special requirements on the intermediate representation and the optimizer are difficult to fulfill on widely available commercial systems. Secondly, Morph does not deal with program source code changes.

A compiler system built at Compaq [Albert 1999] correlates profile information collected for a binary program with the program source code, using help from the compiler back-end. This presents a possible approach to profile propagation, as profile information attached to source code can be carried along when the source is modified. The correlation process in this system is limited by the complex relationship between program source code and binary code. Also, this approach does not address program changes caused by different compilation settings or binary optimizations. Binary matching deals with all types of program changes, and does not require access to source code or additional information from the compiler.

1.5. Major Contributions

This thesis introduces progressive profiling, a new methodology for profile collection in Profile-Based Optimization (PBO). By extending the lifespan of profile information beyond the time when a new version of the program is created, progressive profiling addresses the lack of available profiling time, a serious issue in applying PBO in a software production environment.
This thesis:
- Introduces the methodology of progressive profiling based on binary-level profile propagation and selective profile collection.
- Proposes a set of quantitative metrics for comparing quality of profiles.
- Describes two different binary matching algorithms and measures their performance with profile propagation experiments on popular application programs.
- Implements a complete prototype system based on progressive profiling methodology.
- Designs a number of different models for progressive profiling and evaluates their effectiveness with a case study. Our results show that compared to the current practice, progressive profiling can save a significant amount of profiling time while still maintaining a high-quality profile.

1.6. Outline

In Chapter 2 we discuss the methodology of binary matching and profile propagation in more detail. Before we describe our algorithms and experiments, Chapter 3 summarizes a set of metrics that we use to evaluate our profiles. Two different algorithms for binary matching are explained in Chapter 4 and Chapter 5, each supported by a set of experimental results. Chapter 6 presents our experiments using a prototype system of progressive profiling. We discuss various methodology choices in the process of progressive profiling, and use a concrete example to demonstrate the performance benefits from using different progressive profiling models. Chapter 7 summarizes the thesis.
Chapter 2. Binary Matching and Profile Propagation

This chapter discusses the general methodology of binary matching for the purpose of profile propagation. After briefly reviewing previous work on binary comparison, we define the space of our specific binary matching problem. We then discuss the high-level design of our algorithm as well as the pros and cons of two different approaches to binary matching.

As an essential piece in progressive profiling, profile propagation is the process of converting an existing profile to a new profile that describes a different version of the program. In other words, the existing profile accurately describes how one program version runs, while the new profile needs to predict as accurately as possible how the other version will run. For this purpose we produce a mapping that associates the code in the two versions based on how they will be executed. We associate a block of code in one version with a block of code in the other if they are expected to exhibit the same behavior when the program is used in the same way. Since most profile information is tightly associated with the binary code, our mapping needs to be on the binary level as well.

Binary matching is the process of performing static analysis on two versions of a binary program to generate such a code mapping. For presentation purpose, we define a few terms that we will use throughout this thesis.

Code block refers to a continuous block of code that is a smallest unit to be considered in binary matching. A code block can simply be a basic block. The exact definition of code block may vary for different binary matching algorithms.

A match refers to a link from a code block in one version to a code block in the other version. We use “a–b” to denote a match from code block a to code block b, which indicates that we should propagate profile information from a to b. In some cases the match may be annotated with a calculation that should be applied to the profile information during the propagation. This calculation reflects any predicted change in program behavior due to code changes between the two versions.

In the context of binary matching, we frequently need to refer to a set of two code groups or code representations, one from program version A and the other from version B. For simplicity, we use the term a pair of to describe such a set. For example, a pair of procedures refers to any combination of one procedure from version A and one from version B. Other examples include a pair of code blocks or control tree nodes.

Using the above definitions, we can say that the code mapping produced by binary matching is a set of matches between pairs of code blocks. A match between a pair of code blocks usually indicates an equivalency in execution pattern. Assuming we need to propagate profile information from version A to version B, we take each code block in version B and look for its best equivalent in version A. Ideally, we identify a single matching block in version A so that the two blocks are expected to generate the same profile information under the same workload. If that is impossible due to program changes, we look for one or more fuzzy matches from which we can predict how the code block in version B will behave. Based on this prediction we can generate a formula for calculating new profile information based on the fuzzy matches.

---

4 Execution pattern refers to the runtime behavior of a certain code block under a given workload.
An important by-product of this matching process is our confidence with the matches we find. In the context of progressive profiling, this information will be used to determine the extent of the selective profile collection.

Figure 2.1 shows an example that illustrates the process of binary matching and profile propagation. In this thesis, we use \(a_1, a_2, a_3 \ldots a_n\) to denote code blocks in a procedure in their spatial order. We also use \([a_m : a_n]\) to represent the set of code blocks \(\{a_i, m \leq i \leq n\}\). As we can see from Figure 2.1, profile propagation is a natural extension of binary matching, and the code mapping generated by binary matching directly determines the quality of the propagated profile. Aside from the matching algorithm, the success of binary matching and profile propagation is also affected by how much difference there is between the two program versions. For example, when software developers make significant changes to the program source, the binary code may change so drastically that there is no good way of matching the two versions. The rate of program change may vary for different projects. Even within the same application, some program modules are modified more often or more significantly than the others. The rate of change may also vary for different phases of product development. Late in the development and testing process, there are usually only small changes.

Aside from profile re-use, binary matching and profile propagation have other potential uses that can be explored. For example, if software testers find abnormal behaviors in a new version of a program, binary matching and profile comparison between the new version and the previous version can help programmers identify the possible causes. This method can also be used to evaluate options in program patching.

Figure 2.1. An example of binary matching and profile propagation. The profiles in this example record execution counts for the given code blocks. The code mapping produced by binary matching is used to convert the existing profile for version A to a new profile that describes version B.
2.1. Related Work on Binary Matching

There has been a lot of work on comparing two versions of a binary program or two different binary programs for reasons other than profile propagation. The most common purpose for binary comparison is efficient code patching, such as in a recent project by Baker et al. [1999]. Code patching algorithms generally look for exact or almost-exact matches and deal with a fixed set of simple changes. The goal of profile propagation is much broader. In order to match as much of the program as possible, including substantially altered code sections, our algorithms are designed to accommodate all types of changes from procedure level down to instruction level. Another paper by Baker and Manber [1998] detected similarities in Java bytecodes without referring to the source files, which is similar to our comparing binary programs without using the source code. Their work focused on calculating a single similarity metric for two files, not matching them on code and data block level. Debray et al. [1999] compared a binary program with itself to find redundancies for the purpose of code compression. In addition to finding identical instruction sequences, they detected “similar” basic blocks based on their instruction opcodes, then looked for transformations that would make the blocks identical.

The above binary comparison projects and our work contain similar ideas on looking beyond small changes to find similar codes. However, the ultimate goals are different. In code patching or code compression, the contents of code blocks are the only criterion for binary matching, while the structure of the program or the contents of other code blocks generally do not affect matching decisions. For profile propagation, we match the binaries based on both code contents and program structure. We look at the code blocks not individually but in the context of the binary program, with the ultimate goal of obtaining a propagated profile that accurately describes the new version. For that goal, we may even match two code blocks with very different contents.

As mentioned in Section 1.4, Spike [Cohn et al. 1997] uses binary matching to determine how much of a profile can be re-used. Spike’s binary matching is based a control flow graph signature for each procedure. It uses a minimum edit distance algorithm to find exact matches between the signatures, and propagates the entire profile for each unchanged procedure. If a procedure’s signature changes, existing profile information for that procedure is discarded. Our binary matching and profile propagation occur on a finer level, as we seek to re-use profile information even for procedures that have changed.

2.2. Defining the Problem Space

Since profile propagation is a fairly new research topic, there are many directions to explore. This section discusses several design decisions that limit the scope of our investigation to what we see as the most important cases.

Additional Information that May Help Binary Matching

For the purpose of generality, we assume in our binary matching that there is no additional information other than two versions of a binary program. On the other hand, there are several types of additional information that may be useful for binary matching.

If the differences between two program versions are caused by programmers’ modification of the source code, access to the source code may provide clues for binary
matching. In reality, however, the machine where an application program is installed and executed usually does not have access to the source code. Even in a software production environment, the testing and optimizing of a binary program is often conducted on a separate machine from where the source code is stored. Furthermore, the advantage of source code access is limited by the fact that the association between the source code and the binary code is affected by many code transformations during compilation.

Aside from source code access, the programmers can potentially provide helpful information about the modifications they make to the program. For binary matching purpose, it would be particularly useful if programmers can note any procedure name changes and cross-procedure code movements, such as the combining and splitting of procedures. As we will discuss in Section 2.3, it is difficult to automatically identify these procedure-level changes, especially for large programs with many procedures.

If the differences between two versions are caused by a binary transformation tool such as an optimizer, the tool can potentially provide useful information for binary matching. Alas, it requires a great deal of engineering effort, and the benefit is limited to binary matching between the input and output of the tool.

**Direction of Binary Matching**

Profile propagation is usually directional, going from one version to another but not the reverse. Therefore, our binary matching only considers one direction. In most cases, we have profile for a previously built version and needs profile for a newly built version. We call them the *old version* and the *new version*, respectively. Sometimes we may need to propagate profile information from a later version to an earlier version. In some other cases, the order of creation for the two versions is either unknown or irrelevant. Our binary matching method does not rely on the time relation of two program versions, and our algorithms can be used on any two versions. If we need to propagate profile information in both directions between two versions, we simply consider it as two unidirectional problems. For presentation purpose, the rest of this thesis assumes that we propagate profile information from an old version to a new version. Our code block matches will always point from the old version to the new version.

**Code Matching and Data Matching**

So far we have only discussed code matching. The same method can also be applied to data matching and profile information associated with program data. For a given program, code matching and data matching can be used to help each other. If we perform code matching before data matching, the code matching results can help us match data units whose addresses have changed between two versions. If we perform data matching first, the results will help us analyze code blocks that include data references. Since code-based profiles are more widely used than data-based profiles, we focus on code matching in this thesis. Among our two binary matching algorithms, one deals with both code and data matching while the other deals with code only.

**Speed vs. Accuracy**

A major motivation behind our work is to extend the time available for profile collection in a software production environment where there is time pressure from the daily build cycle. In progressive profiling, even though profile collection no longer has to be conducted entirely within the build cycle, the binary matching and profile propagation process is still a sequential
step in the build process. Therefore, our algorithm should run fairly fast. Instead of doing an exhaustive search for the best matching possibilities, we use a series of heuristic methods to simplify our algorithm while maintaining high matching accuracy. Nonetheless, these heuristic methods may sacrifice some accuracy in the propagated profile.

**Implementation Platform and Porting**

Each of our two binary matching algorithms was implemented on a specific architecture platform. One algorithm was built on the Intel x86 architecture, and the other one on the DEC Alpha architecture. The high-level designs of our algorithms are targeted for general-purpose binary matching regardless of the machine architecture. Therefore, it should be straightforward to port the algorithms to different platforms. However, some choices in the algorithms and the tuning of heuristics are based on the instruction set and characteristics of common application programs on the specific platform. For a different platform, these choices and heuristics may need to be adjusted.

**2.3. Procedure-Level and Block-Level Matching**

Many types of profile information that are gathered today, such as execution counts and branch history, are associated with program basic blocks. For this reason, we set the granularity of our binary matching at the basic block level, even though the code units for our matching algorithm may not exactly be basic blocks. As we have mentioned, we use the phrase code block to refer to the smallest code units considered by our matching algorithm.

An important decision in our algorithm design is to limit the code block matches to within procedure boundaries. We produce a one-to-one mapping on procedure level, and all code blocks in one procedure can only be matched with code blocks in the corresponding procedure in the other version. This restriction narrows down the problem space to focus on intra-procedural changes, which cover most program changes in real cases. Inter-procedural code changes may occasionally occur because of two reasons. Most commercial compilers perform only intra-procedural optimizations, but some compilers perform procedure inlining. In addition, programmers may sometimes split a procedure into several, or combine multiple procedures into one. Due to the large number of procedures in a typical modern application program, these changes are difficult to detect automatically. However, with additional hints from the compiler and programmers, these cases would be relatively easy to handle. In our work, we consider inter-procedural changes a lower priority than intra-procedural changes.

If procedures are deleted from or added to a program, the procedure-level mapping may not cover all procedures in either version. For deleted procedures, their existing profile information will not be used. For added procedures, we will not have propagated profile information for them. In progressive profiling, we can use selective profile collection to generate new profile information for these added procedures.

If programmers never change procedure names, generating the one-to-one procedure mapping is a straightforward task of matching procedures with the same name. In real life, programming languages such as C++ allow name overload, and therefore multiple procedures may have the same name in the source code. However, the compiler will assign each procedure a unique name based on its parameter list and return type.
unfortunately, programmers may change the names of some procedures from version to version. Given that a lot of modern application programs have a large number of procedures and may include many small procedures that are similar to each other, it can be a challenge to find the correct match for a procedure that has undergone a name change. One possible solution is for the programmers to provide information about procedure name changes. Without information from programmers, we can also use heuristics to find procedure pairs that are the same procedure under different names. In our two binary matching algorithms, we use several heuristic methods to deal with procedure name changes. They are based on criteria such as name similarity, signature of code contents, and spatial location in the binary program. We will describe these methods in detail when we present the binary matching algorithms.

2.4. Two Approaches to Block Matching

After procedure-level matching, we take each pair of matched procedures and calculate a mapping between their code blocks. We attempt to find a match for each code block in the new version, either an exact match or a fuzzy match. Since the number of code blocks in a procedure may change from version to version, this block mapping may not be one-to-one. Even if the number of code blocks in a procedure remains the same, the correct mapping still may not be one-to-one due to program changes. If the procedure has changed significantly, our algorithm may not find matches for all code blocks and the mapping may not cover the whole procedure.

When developers modify the source code, they may directly cause several types of changes in the binary program. Such changes include code being added, deleted or moved within a procedure, instruction changes (operands and opcode), and procedure name or type changes. In addition to these direct changes, one small modification in source code may cause many more code and data changes throughout the binary. Examples of these indirect changes include changes in register allocation, changes in instruction scheduling, and changes in control flow instruction targets and data references due to the shift of address space. Indirect changes pose significant challenges for our matching algorithm. In some extreme cases, two blocks that shouldn’t be matched may appear identical while the blocks that should be matched appear different. Figure 2.2 (see next page) illustrates some different types of program changes.

Since the program code can change in many different ways, there can be different approaches to binary matching. In two research projects conducted under different circumstances, we made two different designs.

The first algorithm, based on the code contents of each procedure, aims to match code blocks that correspond to the same part of the source code. It accomplishes this by looking for code blocks with the most similar contents. The underlying assumption is that if two blocks are compiled from the same source code, they will likely behave in the same way when the program is executed. Even though changes in the other parts of the program may change the execution pattern of a certain code block, we assume that when the program changes are small, most code blocks will maintain their execution patterns. If there are significant program changes, we may fail to find matches for some code blocks, and some matches may not produce accurate profile information.

In our second algorithm, we tackled the binary matching problem from a different angle. Since a procedure’s control flow structure dictates the execution pattern within the procedure, we can detect and analyze the changes in execution pattern by comparing the control flow structure
of the two versions. Control flow information alone may not be sufficient, however, as there may be many code areas with the same control flow structure. Therefore, we also use the code contents in our matching algorithm.

Figure 2.3 and Figure 2.4 (see next page) illustrate that code contents and control flow information are both important basis for binary matching, but neither is completely reliable by itself. How to strike a balance between the two was a major challenge in designing our algorithms, and an important topic for continuing research.

There are pros and cons to both code contents-based and control flow-based binary matching. They may be suitable for different situations, depending on the architecture platform and instruction set, the programming language used to write the source code, compiler conventions, and the working style of the programmers. We believe the control flow-based approach is superior in at least two situations:

1. As we already pointed out, a program’s control flow structure directly determines how the code behaves under a certain workload. The code contents-based approach may fail to capture program changes that only affect control flow instructions, while the control flow-based approach focuses on these changes and is therefore more likely to recognize their impact on the program execution patterns.

2. Our code contents-based algorithm is vulnerable to certain types of changes in program code, such as when instructions are added to or deleted from a basic block. Such cases were rare in the benchmark programs for our first project. In the benchmark programs for our second project, however, we noticed a large number of basic blocks where the instruction sequence changes radically from version to version. This may have to do with the fact that the second project was conducted on DEC Alpha, a RISC machine, while the first project was conducted on Intel x86, a CISC architecture.
Figure 2.3. The importance of code contents. In example 1, based on control flow information only, there are two possible ways of matching:

- $a_1 \rightarrow b_1$, $a_2 \rightarrow b_2$, $a_3 \rightarrow [b_3 : b_5]$, $a_4 \rightarrow b_6$
- or $a_1 \rightarrow b_3$, $a_2 \rightarrow b_4$, $a_3 \rightarrow b_5$, $a_4 \rightarrow b_6$ (b1 and b2 are new)

Code contents can help us determine which way is correct.

In example 2, based on control flow information, correct matches appear to be $a_1 \rightarrow b_1$, $a_2 \rightarrow b_2$, $a_3 \rightarrow b_3$, $a_4 \rightarrow b_4$. However, it is possible that the two branches of the if statement have switched place in the program, in which case the correct matches are $a_1 \rightarrow b_1$, $a_3 \rightarrow b_2$, $a_2 \rightarrow b_3$, $a_4 \rightarrow b_4$. The two cases can only be distinguished by code contents.

Figure 2.4. The importance of control flow information. In this example, $a_1$, $a_2$, $a_3$, $a_4$, $a_5$ match with $b_1$, $b_2$, $b_3$, $b_4$, $b_5$, respectively, based on code contents. However, once we look at the control flow structure, we see that the two versions are very different. For the purpose of profile propagation, the matches based on code contents will produce inaccurate profile information.
On the other hand, the control flow-based approach may be less capable under some circumstances:

1. For a code region with complicated control flow structure, such as a loop with many exits and nested sub-loops, certain types of control flow changes may confuse our algorithm and lead to incorrect matches. The code contents-based approach may be able to handle these cases if the non-control-flow instructions remain mostly the same.

2. The control flow-based approach may be affected by a static analyzer’s inability to identify certain control flow information, such as the targets of dynamic jump instructions. The code contents-based approach may have advantage in this case, as it for the most part does not depend on the control flow information. Therefore, it is possible that the control flow-based approach is less suitable for C++ and Java programs than for C programs. Our benchmark programs in this paper are all written in C. We plan to experiment on C++ and Java programs in the future.

Our two projects showed that binary matching is a viable tool on diverse platforms and for variety of program types. Due to time constraints and logistical issues, we have not investigated how each of our two algorithms performs on different platforms, nor have we conducted a direct comparison between the two algorithms on a single platform. How to choose between code contents-based and control flow-based approaches or to combine them is an important topic for future research.

Summary

In progressive profiling, we use binary matching to produce a code mapping that will be used to guide profile propagation. Binary matching also provides confidence information that is important for selective profile collection. When matching two binary programs, we first generate a one-to-one procedure mapping, then conduct code block matching within the procedure pairs. There are different strengths and weaknesses in two approaches to code matching: code contents-based and control flow-based. In Chapter 4 and Chapter 5, we will discuss the details of our two binary matching algorithms.
Chapter 3. Metrics for Profile Comparison

In progressive profiling, we use profile propagation to partly or completely replace profile collection for a new program version. Therefore, we assess the accuracy of the propagated profile by comparing it with a directly collected profile, a profile collected directly on the new version using the same workload that was used to generate the profile for the old version.

How to evaluate and compare the effectiveness of profiles is still an open question. In most PBO research, profiles are evaluated by using them to guide a specific optimization and measuring the performance benefit from the optimization. However, the relationship between a profile and the performance benefit from an optimization guided by that profile is complex, as it depends on the type of optimization, the implementation of the optimizer, and the testing workload. Based on one specific optimization scenario or even several scenarios, it is insufficient to make the general conclusion that one profile is better than another. Furthermore, comparison between optimization performance does not offer much insight to the specific differences between two profiles.

In this thesis, we do not focus on the problem of evaluating profiles by their optimization performance. Instead, we use quantitative methods to conduct detailed statistical comparison between profiles. We always assume that the directly collected profile is accurate for a specific program version and a given training workload, even though it may not be the optimal profile for all optimization scenarios. A propagated profile is accurate if it is exactly the same as the directly collected profile for the new version. The smaller the differences between these two profiles, the more accurate and “better” we say the propagated profile is. Based on this definition, we further define a correct match in binary matching as a match that will produce accurate profile information.

The approach of using statistical comparison instead of optimization-based comparison was investigated in one of our previous projects on PBO [Gloy et al. 1997]. In that work, we used profiles produced by time-based sampling to replace complete profiles collected through instrumentation. In terms of methodology, it was similar to using propagated profiles to replace directly collected profiles. From the experimental results in that project we can make the following observations:

1. In most cases, the complete profile offers the best optimization performance. However, some sampling profiles achieve better performance than the complete profile. Note that we used different workloads for training and testing.

2. We used two different optimization-based measurements: cache miss rate and program execution time. In some cases, these two measurements lead to opposite conclusions in comparing the effectiveness of the same two profiles.

3. For most benchmarks, the 32-run sampling profile, which is statistically more alike the complete profile, achieves better optimization performance than the 1-run sampling profile.

The above observations suggest that optimization performance may not be a good method for general comparison between profiles, and that statistical comparison can serve as a good indicator for the profiles’ potential for optimization performance.
Statistical comparison between profiles is also useful in other areas of PBO. For example, when applying PBO on real users’ machines, we need to compare collected profiles in order to detect changes in program behavior and trigger the optimizations.

In this chapter we present a list of metrics for profile comparison. Summarized under three categories, these metrics will be used in later chapters to evaluate the results of profile propagation. Some of them are selected from existing work, while others are by our own design. Depending on specific purposes, different metrics may be suitable for different cases of profile comparison.

All the metrics discussed in this chapter can be applied to the whole profile or just a subset of the profile. For example, we can perform profile comparison for each procedure individually and thus measure the differences between profiles on a finer level.

### 3.1. Optimization-Specific Comparison

Given two profiles \(P\) and \(Q\), a profile-based optimizer \(OPT\), and a performance metric \(PERF\), an optimization-specific comparison between the two profiles can be defined as the following:

\[
RATIO_{OPT, PERF}(P, Q) = \frac{PERF(OPT(P))}{PERF(OPT(Q))}
\]

This method compares two profiles in terms of how effective they are for a specific optimization, as measured by a specific performance metric. It does not reflect the accuracy of any information in the profiles that does not affect the given optimization or performance metric.

Though many research projects have used results from a specific optimization to compare the quality of two profiles, the primary motivation of those projects was to identify improvements in optimization performance. Since optimization performance is not our main criterion for profile quality in this thesis, we include only one optimization-specific metric.

### 3.2. Statistical Methods of Comparison

In contrast with the optimization-specific approach, statistical methods look at two profiles from a pure mathematical point of view and aim to quantify their similarity, regardless of what purpose they will be used for later. Without losing generality, we view a profile as a discrete function on a certain type of program entity. In this model a profile \(P\) is a collection of pairs:

\[
P = \{ (x_i, f_P(x_i)) \mid 1 \leq i \leq N \}
\]

Here \(x_i\) is a profiling unit, such as a code area or a point in the program, and \(f_P(x_i)\) is the profile value for \(x_i\). For example, \(x_i\) may be a basic block and \(f_P(x_i)\) is the execution count for that basic block. In our discussion we assume \(f_P(x_i) \geq 0\), which is true for most types of profile information. \(N\) is the total number of profiling units in the profile.

Various statistical methods have been used to measure the difference between two profiles. In the earlier project by Gloy et al. [1997] we used the \(\chi^2\) goodness-of-fit test [Lindgren 1976], which compares one set event distribution frequencies against another. Savari and Young [1999] employed a similar approach, using the relative entropy of Kullback-Liebler distance to measure the distance between an assumed profile and an actual profile. One shortcoming of these
tests is that their results are not intuitive. Both $\chi^2$ test and relative entropy test calculate a sum over all profiling units, so their results may grow indefinitely large especially when the profiles are large. In a general context, the similarity between two profiles is not immediately clear from their $\chi^2$ or relative entropy results. Another characteristic of these tests is that they are not symmetric, and in fact may produce very different values if the two profiles are switched.

A metric proposed by Kistler and Franz [1998] regards each profile as an $N$-dimension vector formed by its $N$ profile values and uses the geometric angle between the two vectors to measure the similarity between two profiles. This test is symmetric and always produces a result between $0^\circ$ and $90^\circ$. However, an angle in an $N$-dimension space is still not an intuitive concept when $N$ is large.

To address the problems with these sophisticated tests, we propose a set of simple metrics that can intuitively demonstrate how different two profiles are.

Given two profiles $P$ and $Q$, we can do a simple comparison by counting the percent of profiling units that have the same profile value. We call this metric the Equal Value Percentage, or simply E.V. Percentage:

$$E.V. \text{ Percentage} (P, Q) = \frac{\# \{ i | (1 \leq i \leq N) \land (f_P(x_i) = f_Q(x_i)) \}}{N}$$

Obviously, E.V. Percentage is 100% if and only if the two profiles are strictly equal. A less strict measure is the percent of profiling units that have similar profile values. In order to define “similar”, we first define a function that calculates the relative difference between two values:

$$\text{Max-to-Min} (p, q) = \begin{cases} \frac{(p + \delta)}{(q + \delta)}, & \text{if } p > q \\ \frac{(q + \delta)}{(p + \delta)}, & \text{if } p \leq q \end{cases}$$

where $\delta$ is a pre-selected small value to avoid dividing by zero. The value of $\delta$ also determines how much we penalize the difference between a zero and a non-zero profile value. In our experiments $\delta$ is 0.001, which means $\text{Max-to-Min} (0, 1) = 1000$. Using this function we define the metric Similar Value percentage, or S.V. percentage:

$$S.V. \text{ Percentage} (P, Q) = \frac{\# \{ i | (1 \leq i \leq N) \land (\text{Max-to-Min}(f_P(x_i), f_Q(x_i)) \leq T) \}}{N}$$

where $T$ is a constant that defines the similarity threshold, and $T > 1.0$. For example, if $T$ is 1.5, S.V. Percentage gives us the percentage of profiling units for which the two profiles provide profile values no more than 50% apart (assuming a very small $\delta$). If $T$ is set to 1.0, S.V. Percentage becomes equivalent to E.V. Percentage.

E.V. Percentage and S.V. Percentage indicate the proportion of the two profiles that are identical or similar, and therefore are suitable for evaluating the success rate of profile propagation. They do not measure how different the other parts of the profiles are. To quantify the overall difference between two profiles, we use a metric called Max-to-Min Average, or M.-M. Average, which calculates the geometrical mean of the Max-to-Min function over all $N$ profiling units.

$$M.-M. \text{ Average} (P, Q) = \left( \prod_{i=1}^{N} \text{Max-to-Min}(f_P(x_i), f_Q(x_i)) \right)^{1/N}$$
By using the geometrical mean, this metric measures the difference in terms of the order of magnitude of profile values. \textit{M.-M. Average} is always greater than or equal to 1.0, and is equal to 1.0 if and only if \( P \) and \( Q \) are entirely equal. It tells us how different the profile values are as averaged over all profiling units. Two pairs of profiles may have the same \textit{E.V. Percentage} and \textit{S.V. Percentage} but different \textit{M.-M. Average}. On the other hand, two profiles with mostly equal and a few very different profile values may produce the same \textit{M.-M. Average} as two profiles with mostly different profile values.

There is an existing metric called \textit{Coverage} [Feller 1998] that also summarizes the numerical similarity of all profile values. The definition is simple:

\[
\text{Coverage} (P, Q) = \frac{\sum_{i=1}^{N} \min(f_P(x_i), f_Q(x_i))}{\sum_{j=1}^{N} \max(f_P(x_j), f_Q(x_j))}
\]

In this thesis we choose to use \textit{M.-M. Average} instead of \textit{Coverage}. As is often the case with execution count profiles, a profile may contain very large profile values as well as very small ones. In this case, \textit{Coverage} is mainly determined by any changes to the large values, and therefore does not reflect the overall effectiveness of our profile propagation. \textit{M.-M. Average}, on the other hand, measures the accuracy of the small profile values as well as the large ones.

One potential problem with the three metrics we proposed above is that they are sensitive to the scale of profile values. Sometimes, profile \( P \) has the same “shape” as profile \( Q \) but is on a different scale, i.e.,

\[
f_P(x_i) = f_Q(x_i) \times C, \quad 1 \leq i \leq N, \text{ where } C \text{ is a constant}
\]

This often occurs because basic block execution counts in a procedure are proportional to the execution count of that procedure. If the procedure is executed for twice as many times under the same condition, all the basic block execution counts also double. In this case, \textit{E.V. Percentage} and \textit{M.-M. Average} will imply that \( P \) and \( Q \) are very different, and so will \textit{S.V. Percentage} when \( C \) is sufficiently larger or smaller than 1.0. In reality, however, most intra-procedural optimizations are insensitive to the scale of profile values, and would yield the same results from profiles \( P \) and \( Q \). Therefore, it is more practical to consider \( P \) and \( Q \) as equivalent profiles.

To address this problem, we normalize all execution counts before we compare the profiles. We divide all basic block execution counts in a procedure by the execution count for the first basic block, which for a single-entry procedure is the execution count for the procedure. This normalization process produces execution counts that are floating point numbers, some of which may be less than one. This, however, does not affect the calculation of our metrics.

### 3.3. Hot Code Tests

The metrics in this category do not simply regard a profile as a set of numbers; they also consider the significance of the profile values in optimizations. Many profile-based optimizers use execution count profiles to pinpoint the most frequently executed areas of the program, and then apply optimizations on those areas. Examples of this include code layout [Pettis and Hansen 1990] and profile-based path selection [Young 1998]. For these optimizers, the exact profile values are not important, but their relative rank is. If two profiles are numerically different but indicate the same set of frequently executed code, they will be equally effective for these
optimizations. Here we define two metrics that compare execution count profiles based on their hot code selections.

The first metric simply regards all profiling units with non-zero execution counts as “hot”. In other words, we compare the set of code that, according to each profile, is touched upon during the execution. Given two profiles, each profiling unit is labeled “agreed” if it has non-zero execution count in both profiles or zero count in both. The Correct Coverage Percentage (C.C. Percentage) is defined as the percentage of “agreed” profiling units.

To compare only the most frequently executed code, we use the Key Matching method designed by Wall [1990]. For an integer \( n, 1 \leq n \leq N \), the key matching function \( K.M.(n) \) is the number of profiling units that rank among the top \( n \) most frequently executed entries in both profiles. Obviously we have \( K.M.(n) \leq n \) and \( K.M.(n+1) \geq K.M.(n) \). The closer \( K.M.(n) \) is to the identical function, the more similar the two profiles are in terms of hot code selection. \( K.M.(n) \) is equal to the identical function if and only if the two profiles rank all profiling units in the exact same order.

The definition of the key matching function is ambiguous when there are ties in profile values. In our experiments, we rank the profiling units first by execution count then by the order they appear in the program, and thus remove the ambiguity. Figure 3.1 illustrates the key matching function with an example.

<table>
<thead>
<tr>
<th>Profile P</th>
<th>Profile Q</th>
<th>Key Matching</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>sorted</td>
<td>original</td>
</tr>
<tr>
<td>a1  1</td>
<td>1 a3</td>
<td>a1  1</td>
</tr>
<tr>
<td>a2  0</td>
<td>2 a7</td>
<td>a2  1</td>
</tr>
<tr>
<td>a3 100</td>
<td>3 a4</td>
<td>a3 200</td>
</tr>
<tr>
<td>a4 80</td>
<td>4 a5</td>
<td>a4 90</td>
</tr>
<tr>
<td>a5 80</td>
<td>5 a9</td>
<td>a5 90</td>
</tr>
<tr>
<td>a6 20</td>
<td>6 a6</td>
<td>a6 110</td>
</tr>
<tr>
<td>a7 100</td>
<td>7 a1</td>
<td>a7 200</td>
</tr>
<tr>
<td>a8 1</td>
<td>8 a8</td>
<td>a8 1</td>
</tr>
<tr>
<td>a9 50</td>
<td>9 a10</td>
<td>a9 50</td>
</tr>
<tr>
<td>a10 1</td>
<td>10 a2</td>
<td>a10 1</td>
</tr>
</tbody>
</table>

Figure 3.1. Illustration of the Key Matching function.
Summary

To evaluate the quality of a propagated profile, we compare it with a profile collected directly on the new version. This chapter summarizes three types of metrics for profile comparison: optimization-specific metrics, statistical methods, and hot code tests. Since there are many variables involved in using optimization performance to evaluate profile quality, we instead focus on using quantitative methods to conduct statistical comparison of two profiles.
Chapter 4. Code Contents-based Matching Algorithm

We first used binary matching to explore profile re-use in a research project conducted at Microsoft Research [Wang et al. 2000]. Our project was motivated by the difficulty of collecting enough profile information for large applications under the restriction of a daily build cycle. We developed BMAT, a binary matching tool, on the Intel x86 architecture under Windows NT. BMAT deals with both code matching and data matching. It uses Vulcan, a binary analysis tool built by Srivastava et al. [1999], to create an intermediate representation of x86 binaries, which frees us from the tasks of separating code from data and identifying program symbols.

In Section 2.4 we discussed two types of binary matching algorithm, i.e., code contents-based and control flow-based. The binary matching algorithm in BMAT is mostly based on code contents, and uses control flow information only for certain unresolved cases. This design assumes that most changes between the two program versions are minor and local. Therefore, we can find correct matches by looking for code blocks with similar contents. Another assumption is that changes in program control flow structure are rare, and thus most code blocks maintain their execution patterns from version to version. By disregarding control flow changes, however, we do run the risk of generating inaccurate profile information under some circumstances.

In BMAT, our code blocks (i.e. the smallest units for code matching) are simply program basic blocks. We use a hashing method to compare basic blocks based on their contents. A 64-bit hash value is calculated for each basic block based on the opcodes and operands of its instructions. If a pair of basic blocks have the same hash value, it is likely that they constitute a match. If two or more basic blocks in one procedure have the same hash value, we use locality information and heuristic methods to identify which one makes the best match. The use of locality information is based on the assumption that the order of basic blocks in a procedure seldom changes.

The hashing calculation in our implementation is order-sensitive. In other words, two basic blocks produce the same hash value if and only if their instructions match and the order is the same. If aggressive instruction scheduling is performed during compilation, changes in one block may cause instructions in other blocks to shuffle around. In that case, it may be desirable to use a hashing method that does not depend on the order of the instructions. This issue is not investigated in this thesis.

In binary matching, we need to find corresponding blocks in two versions despite many possible types of differences they may have. This introduces a tradeoff between being flexible to look past minor changes and being precise to identify correct matches. As we pointed out in Section 2.4, even code that remains the same in the source code may undergo indirect changes. However, if we ignore all program elements that may be affected by indirect changes, we will have too little information left to find any matches. In our design, the hashing-based matching algorithm includes multiple passes with different levels of “fuzziness”. This allows us to find more correct matches. A number of heuristic methods are used to help find the best match for each block.

As discussed in Section 2.3, our binary matching process is divided into two stages. The first stage is to find a one-to-one mapping between the procedures in two versions. Procedures

---

6 This chapter is revised from part of a previously published paper by Wang et al. [2000].
that have been deleted or added will not be included. During the second stage, we look for basic block matches within each pair of matched procedures. Data matching is conducted in a separate phase from code matching. We chose to perform data matching before code matching in order to use data matching results to aid code matching. To keep our algorithm simple and fast, we decided not to investigate the possibility of doing data block matching in multiple passes or intertwining the two.

Figure 4.1 is an overview of the entire matching process. In the rest of this chapter, Section 4.1 presents the general design of procedure matching, data matching and intra-procedural code matching, while Section 4.2 discusses some details of our x86 implementation. Section 4.3 presents the experimental results.

### 4.1. Algorithm Design

#### 4.1.1. Procedure Matching

In the procedure matching stage, we use a combination of name-based and contents-based methods to generate a one-to-one mapping between the procedures in two program versions.

Our most basic idea is that if two procedures have identical names, we map them to each other. When procedure overloading is allowed, multiple procedures may have the same *hierarchical name*\(^7\). However, the compiler assigns each procedure a unique *extended name*, which includes not only the hierarchical name but also the procedure parameters and return type information. In our algorithm, we first look for procedure pairs with the same extended name. Then, among the remaining procedures, we look for pairs with the same hierarchical name. Both steps are necessary because when the parameter list or return type of a procedure changes, the extended name changes even though the hierarchical name remains the same.

---

\(^7\) For C++ programs, we consider the class hierarchy as part of the procedure name. For example, if *foo* is a class and it has a method *bar*, the corresponding procedure name will be *foo::bar* instead of just *bar*. We refer to such a name as the procedure’s *hierarchical name*. 

---
Occasionally program developers change the name of a procedure. We have observed that in most cases, the change involves only a small number of characters in the name string. Therefore, we look for every procedure pair whose hierarchical names are different by a small number of characters, then do a block trial-matching between their code blocks. This trial-matching is a simplified, one-pass version of the contents-based basic block matching algorithm performed later within each pair of matched procedures. If the percentage of matching blocks between the two procedures is high, we conclude that the two constitute a match. The thresholds for procedure name difference and block matching percentage are both set heuristically, based on observations on some common Windows NT applications. Particularly, the threshold for procedure name difference may need to be adjusted according to the habits of program developers.

For procedures that cannot be matched by name, we can also perform pair-wise comparison based on their code contents. This is a coarser comparison than the block trial-matching because it uses one single value to represent each procedure’s entire code contents. In a bottom-up fashion, we calculate a single hash value for each procedure based on the hash values of its basic blocks. The calculation is sensitive to the order of blocks within a procedure. We then compare the procedures’ hash values to look for matches. Like in the basic block matching phase, this hashing-and-comparing process is done in multiple passes with different levels of fuzziness.

If there are still procedures remaining unmatched, they are usually procedures that have been deleted or added. Some of them may be procedures whose name and contents have both changed. To catch the latter cases, we perform the above-mentioned block trial-matching between all unmatched procedures. Unlike procedure comparison based on entire code contents, block trial-matching can find a match even when code blocks have been added to or deleted from a procedure. We have observed that for many programs, this method can be time-consuming and yet find very few procedure matches. In a commercial situation with time pressure, it might be desirable to skip this step.

In our implementation of BMAT, the above four methods are performed in the following order: matching by identical names, matching by entire code contents, matching by similar names, and matching by block trial-matching. We put matching by similar names after matching by entire code contents because the former has a high error rate for some modern applications where many procedures have similar names.

4.1.2. Data Matching

We divide the program data sections into data blocks based on their access pattern. Each data address that is referred to explicitly starts a new data block. As with code matching, we use a hashing algorithm to match data blocks based on the values stored in each block. Relocation entries pose challenges because their values tend to change from build to build. In our algorithm we exclude all relocation entries from the hashing, although more sophisticated solutions are possible for this problem. For data blocks that are not matched by value, we try to match them according to their positions in the program. If a pair of unmatched data blocks are sandwiched by two pairs of matched data blocks in the program, we make them a match as long as they have similar sizes.
4.1.3. Code Matching

Within each pair of matched procedures, we use a hashing algorithm to match the basic blocks based on their code contents, taking their relative positions into account. We perform multiple hashing passes with different levels of fuzziness. Details of the hashing and comparing process will be described in Section 4.2.

The contents-based matching identifies only one-to-one matches between basic blocks. It cannot find matches for blocks that have been deleted, added or drastically changed. In order to propagate as much profile information as possible, we try to match each of these remaining blocks with a block that is equivalent according to the control flow. Specifically, we traverse down both versions of a procedure simultaneously following the control flow, and use conditional branches, jump instructions, return instructions and previously matched blocks to pinpoint code sections that are comparable in terms of control flow. We do not use subroutine calls as reference points because they are often added to, removed from or moved within a procedure while the rest of the code remains the same. This traversing method can find a match for every block except those that are unreachable in the scope of our static analysis. Unlike contents-based matching, the control flow-based phase may match several blocks in the same control flow branch to a single block in the other version.

![Diagram of code matching](image)

**Figure 4.2. Example of the control flow-based matching phase.** In this example, two new blocks (b2’ and b2”) are added to the fall through path of a conditional branch. During the control flow-based matching phase, these two blocks are matched to a2, the fall through block of the same branch in the old version. Note that b2, b2’ and b2” are matched to the same block, but each new block is still matched to only one old block.

A “partial” flag is attached to the match from block a2 to block b2”, indicating that b2” corresponds to only part of the control flow through a2. In other words, if a2 is executed N times under a certain scenario, b2” may be executed anywhere between 0 and N times under the same scenario. In our implementation, we estimate the count for b2” as $N/2$.

4.2. Implementation Details

This section discusses some details of the multi-pass hashing-and-comparing process in both procedure matching and basic block matching, and explains the reasons behind some of our design decisions.
4.2.1. Types of Code Changes

In order to match as many basic blocks as possible and match them correctly, we need to filter out indirect changes and also accommodate minor direct changes. We achieve this by excluding certain information from the hashing calculation. Here is a list of different types of information we may need to exclude in order to find correct matches:

- **Numerical address offsets**, i.e., numerical offsets in memory address operands. These offsets often change from build to build due to changes in data layout.

- **Register allocation.** A minor code change may cause different results of register allocation for the rest of the procedure. This only affects registers that are included in the allocation by the compiler. It does not affect special registers such as the stack pointer.

- **Instruction(s) added or removed.** One or more instructions may be added to or removed from a basic block, and each instruction may change so much that our hashing algorithm cannot accommodate the changes. To match such a block requires the exclusion of these instructions. It is impractical to try out all the possibilities, as the total number of possibilities is exponential with the number of instructions in the block. For simplicity, our design is to hash only the last instruction in the block during some passes.

- **Instruction opcode and operand types.** For example, due to minor changes in the source code or the compilation process, a *push word* instruction may become a *push double word*, a memory operand may become a register operand, and a *return* instruction with no parameter may become a *return* with a parameter.

- **Immediate operands.** Some immediate operands, such as loop boundaries and program constants, may change from build to build.

- **Block address operands.** These operands appear in control flow instructions (jump, branch and call) and some others such as pointer operations. For simplicity, we call the instruction that contains the block address operand *source instruction* and the block referred to by the operand *target block*. In many programs, the majority of indirect changes occur to block address operands, and these changes can be tricky to recognize. It is necessary to distinguish the following scenarios:
  
  - The source instruction is modified to refer to a different target block. This is a direct change to the block address operand.
  
  - The target block address changes, which could be caused by the shifting of the whole procedure or the shifting of the block within the procedure. This is an indirect change to the block address operand.
  
  - The target block code changes, or the target procedure’s name or type is modified. This does not affect the block address operand directly, but it may be confused with the first scenario.

With all these possibilities, the best way to represent a block address operand is not clear. Our design includes the following rules:

1. If the target block is already matched to another block, the hashing is done so that address operands associated with the two blocks are treated as the same.

2. If the target block has a compiler-assigned extended name (for example, if it is the entrance block of a procedure), hash the extended name only.
3. If the target block is within the same procedure as the source instruction, hash the address offsets of the target block from the beginning of the procedure and from the source instruction. If the target block is in a different procedure, hash the name of that procedure and the address offset of the target block from the beginning of that procedure.

We follow these rules in the given priority order. However, steps 1 and 2 may introduce errors in some cases. Therefore, they are disabled in some passes during the matching process, as described later in Section 4.2.2.

In summary, there are many types of information we may need to exclude in order to find matches. Instead of trying out all the combinations among them, we selectively define several matching fuzziness levels, and perform multiple matching passes at different levels. At each level, we exclude a specific set of information from the hashing calculation. The design is to have levels where most information is included in the hashing, and also levels where much information is excluded. In other words, we use different degrees of approximation when we look for matches at different levels. The more encompassing levels allow us to find accurate matches for blocks that have not changed or only barely changed, while the fuzzier levels find good matches for blocks that have changed considerably.

The definition of the fuzziness levels and the order in which we perform them are both heuristic. We tuned the heuristics based on our experiments on several common Windows NT applications. The tuning often involves tradeoffs between matching accuracy and algorithm complexity.

### 4.2.2. Definition of Matching Fuzziness Levels

Here we list the definition of all matching fuzziness levels in our implementation of BMAT. Generally speaking, the levels are defined incrementally, i.e., more and more information is excluded when the fuzziness level increases. There are two special levels where we consider only the last instruction in each block. They are designed to deal with blocks in which instructions have been added or removed. Table 4.1 (see next page) gives an overview of all fuzziness levels. Next we explain each level in more detail.

- **Level 0.** All instruction opcodes and operands are included in the hashing. Due to indirect changes in address offsets and register allocation, a level 0 match between two blocks is not necessarily a correct match. In our current implementation, level 0 is not used in basic block matching. It is used in procedure matching to find procedures that remain exactly the same.

- **Level 1.** At this level and above, numerical address offsets are excluded from the hashing, and registers EAX, ECX and EDX are converted to the same value for the calculation. These three registers are allocated for arithmetic calculations by the compiler that generated our benchmark programs. We retain some information on the use of these three registers by including their dependencies within each basic block in the hashing. Figure 4.3 (see next page) provides an example.

- **Level 1a.** Same as level 1, but hashes only the last instruction in each block.

- **Level 2.** For each block address operand, the address offset of the target block from the beginning of the procedure is excluded. This accommodates indirect changes that cause address shift for part of a procedure.
<table>
<thead>
<tr>
<th>Fuzziness</th>
<th>Numerical Address</th>
<th>Register Allocation</th>
<th>Block Address Operand</th>
<th>Operand</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>all</td>
<td>all</td>
<td>• target block’s match&lt;br&gt;• target block’s extended name&lt;br&gt;• target procedure name or branch&lt;br&gt;offset within procedure&lt;br&gt;• target block’s distance from&lt;br&gt;beginning of procedure</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>1</td>
<td>none</td>
<td>EAX/ECX/EDX: dependency only; others: all</td>
<td>same as level 0</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>1a</td>
<td>same as level 1, but includes only the last instruction in each block</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>none</td>
<td>EAX/ECX/EDX: dependency only; others: all</td>
<td>• target block’s match&lt;br&gt;• target block’s extended name&lt;br&gt;• target procedure name or branch&lt;br&gt;offset within procedure</td>
<td>all</td>
<td>all</td>
</tr>
<tr>
<td>3</td>
<td>none</td>
<td>EAX=ECX=EDX; EBX=EDI=ESI; others: all</td>
<td>target procedure name or&lt;br&gt;branch direction within procedure</td>
<td>no immediate; none for return</td>
<td>all</td>
</tr>
<tr>
<td>3a</td>
<td>same as level 3, but includes only the last instruction in each block</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>none</td>
<td>N/A</td>
<td>type only</td>
<td>all</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>none</td>
<td>N/A</td>
<td>none</td>
<td>group</td>
<td></td>
</tr>
</tbody>
</table>

**Table 4.1. Definition of matching fuzziness levels.** For each fuzziness level, this table lists how much information is included in the hashing calculation.

---

**Figure 4.3. Excluding register allocation and including register dependency.** By treating EAX, ECX and EDX as the same in the hashing calculation, we can match blocks (1) and (2) despite indirect changes in register allocation. Meanwhile, by including register dependency information, we can still distinguish blocks (1) and (2) from (3).
Level 3. At this level:
- All immediate operands and operands of return are excluded from the hashing.
- In addition to EAX/ECX/EDX, registers EBX/EDI/ESI are also converted to the same value for the hashing calculation. These three registers are usually used as base registers for memory access. Register dependency information is no longer included, and thus the registers in each of the two groups make no difference to the final hash value.
- For block address operands, matching status and extended name of target blocks are no longer used. Address offset from a source instruction to a target block in the same procedure is reduced to +1 or –1 based on the branch direction (forward or backward).

Level 3a. Same as level 3, but hashes only the last instruction in each block.

Level 4. For each instruction, hash the opcode and types (not contents) of operands.

Level 5. For each instruction, hash the opcode only. Certain groups of opcodes are considered as the same, such as push word and push double word, all conditional branch opcodes, etc.

4.2.3. Method of Matching at Each Level

At each matching fuzziness level, there can be several code units (procedures or blocks) with the same hash value. The possibility of ties tends to increase with higher fuzziness levels as we exclude more information from the hashing. We refer to a pair of code units with the same hash value as matching candidates.

For procedure matching, ties are rare except for procedures with only a few basic blocks. As a heuristic, we break the ties by matching the candidates in their order of appearance in the program. At the higher fuzziness levels, we ignore very small procedures to reduce the number of ties, and also use block trial-matching (see Section 4.1.1) to help avoid wrong matches.

For block matching within a pair of procedures, ties are relatively common. We use a two-phase method to break the ties, utilizing a locality assumption that the order of blocks in a procedure usually does not change drastically. This assumption may not hold true if aggressive basic block ordering is performed during compilation.

In the first phase, the one-to-one phase, we match blocks that are the only matching candidate for each other. This is reasonable for fuzziness level 1, but at higher levels, even a one-to-one candidate does not necessarily make a correct match. Errors are especially likely when many small blocks are similar to each other. Most of these errors fall into the category of cross-matching, where a new match “crosses” an existing match (see Figure 4.4). Based on our observations and experiments, we added the following restrictions: At fuzziness level 1a, cross-matching is forbidden. At level 3, cross-matching is forbidden for blocks with three instructions or less. At level 3a, we skip the one-to-one phase altogether. At levels 4 and 5, we forbid any matching for blocks with one or two instructions, and forbid cross-matching for blocks with three instructions.

Figure 4.4. An example of cross-matching.

\[
\begin{array}{|c|c|}
\hline
\text{a1} & \text{b1} \\
\text{a2} & \text{b2} \\
\text{a3} & \text{b3} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
\text{a1} \\
\text{a2} \\
\text{a3} \\
\hline
\end{array}
\]

\[
\begin{array}{|c|}
\hline
\text{b1} \\
\text{b2} \\
\text{b3} \\
\text{b4} \\
\hline
\end{array}
\]

\[\rightarrow: \text{an existing match}\]
\[\leftrightarrow: \text{a new match}\]
The second phase is the propagation phase, where we propagate a match between a pair of blocks to their neighbor blocks. To implement this, we look through the multiple matching candidates for a block and pick one that passes the neighbor test. A pair of blocks passes the neighbor test if either their predecessors or their successors are matched to each other. This new match can then be propagated to more successors or predecessors. This process is repeated until the propagation stops.

4.2.4. Order of Matching Passes

We can adjust the order in which different levels of matching are performed. Note that not all levels have to be used, and each level can be used more than once. In general, we perform the matching levels in increasing order of fuzziness. However, we have observed that for certain cases, a strictly increasing order is not the best choice. We adjusted the order accordingly. In our current implementation of contents-based procedure matching, we perform six passes using fuzziness levels of 0, 1, 3, 1a, 5, and 3a, in that order. Levels 2 and 4 are skipped in an effort to save time, as they do not significantly improve matching results on top of the other passes. During basic block matching, we perform seven passes using fuzziness levels 1, 3, 2, 1a, 4, 3a, and 5. Level 0 is skipped because it may cause wrong matches.

An annotation is attached to each code block match to indicate the fuzziness level at which the match is established. It is also used to mark matches that are made during control flow-based block matching. This annotation helps to distinguish between strong and weak matches, and thus provides important information for the profile propagation algorithm.

4.2.5. An Example

To demonstrate our basic block matching algorithm, Figure 4.5 shows the matching results for a procedure from a common Windows NT application.

<table>
<thead>
<tr>
<th>Version 1</th>
<th>Version 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov eax,[Sym_a1]</td>
<td>mov eax,[Sym_b1]</td>
</tr>
<tr>
<td>test eax,eax</td>
<td>test eax,eax</td>
</tr>
<tr>
<td>je L1 (offset: 0x30)</td>
<td>je L1 (offset: 0x2B)</td>
</tr>
<tr>
<td>test byte ptr [Sym_a2+3Eh],10h</td>
<td>test byte ptr [Sym_b2+3Eh],10h</td>
</tr>
<tr>
<td>je L1 (offset: 0x27)</td>
<td>je L1 (offset: 0x22)</td>
</tr>
<tr>
<td>push 1</td>
<td>push 1</td>
</tr>
<tr>
<td>push dword ptr [Sym_a2]</td>
<td>push dword ptr [Sym_b2]</td>
</tr>
<tr>
<td>push 453h</td>
<td>push 453h</td>
</tr>
<tr>
<td>push eax</td>
<td>push eax</td>
</tr>
<tr>
<td>call dword ptr [Sym_a3]</td>
<td>call dword ptr [Sym_b3]</td>
</tr>
<tr>
<td>and dword ptr [Sym_a4],0</td>
<td>and dword ptr [Sym_b4],0</td>
</tr>
<tr>
<td>and dword ptr [Sym_a5],0</td>
<td>and dword ptr [Sym_b5],0</td>
</tr>
<tr>
<td>jmp Proc_a6</td>
<td></td>
</tr>
</tbody>
</table>

L1: ret

Figure 4.5. An example of code matching. The number on each arrow is the annotation that indicates the fuzziness level at which the match is made. CF = control flow-based.
In Figure 4.5, the data symbols are already identified in the intermediate representation, and Sym_a1, Sym_a2, Sym_a3, Sym_a4, Sym_a5 have been matched with Sym_b1, Sym_b2, Sym_b3, Sym_b4, Sym_b5, respectively, during data block matching. Assuming the data symbols were not identified or not matched, the algorithm would still find the same code block matches, but possibly at higher fuzziness levels.

The different jump offsets for the first two pairs of blocks are indirect changes. These two pairs are matched at level 3 when jump offsets are reduced to jump directions. For the fourth pair of blocks, the last instruction has been removed in the new version, so the blocks are not matched until control flow-based matching.

4.3. Experimental Results

In this section we present experimental results for mshtml.dll, a major DLL from Microsoft Internet Explorer 5.0, and a collection of eight DLLs from Microsoft Windows 2000. When tuning the heuristics in our algorithm, we used two of the Windows 2000 DLLs and another small program, but none of the other DLLs.

4.3.1. Running Time

Table 4.2 lists the running time of BMAT for some programs of different sizes. All experiments were run on a PC with a Pentium II 200MHz processor and 512MB of RAM. The total running time ranges from four seconds for a 122KB-sized program to under four minutes for a 5.5MB-sized program. For these programs, between a quarter and a half of the total running time is spent in the Vulcan tool [Srivastava et al. 1999] to build the program intermediate representation, while the rest is spent by the matching algorithm.

<table>
<thead>
<tr>
<th>Weeks Apart</th>
<th>File Name</th>
<th>File Size (KB)</th>
<th>Running Time (seconds)</th>
<th>Speed (Minute / MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Build</td>
<td>Match</td>
</tr>
<tr>
<td>3</td>
<td>dhcpcsvc.dll</td>
<td>122</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>netlogon.dll</td>
<td>477</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>browseui.dll</td>
<td>1080</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>shell32.dll</td>
<td>3360</td>
<td>32</td>
<td>84</td>
</tr>
<tr>
<td>21</td>
<td>mshtml.dll</td>
<td>5628</td>
<td>60</td>
<td>178</td>
</tr>
</tbody>
</table>

Table 4.2. Running time of BMAT.

Weeks Apart: Time apart between the two builds on which we ran BMAT
Build: Time for building intermediate representations using the Vulcan tool
Match: Time for the matching algorithm
Total: Total time for building and matching
Speed: Total running time divided by file size
4.3.2. Matching Rate

Table 4.3 lists the matching rate results for mshtml.dll. These results may not necessarily prove the accuracy of our algorithm, as they are also affected by the amount of difference between two program versions. As shown in the table, the matching rate indeed decreases when the builds are longer apart in time. The table also shows that mshtml.dll did not change drastically over several months of time, a positive sign for the potential of profile propagation.

<table>
<thead>
<tr>
<th>Build Date</th>
<th>10/22/98</th>
<th>03/15/99</th>
<th>04/16/99</th>
<th>07/15/99</th>
<th>07/27/99</th>
<th>08/11/99</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Size</td>
<td>5395KB</td>
<td>5592KB</td>
<td>5592KB</td>
<td>5600KB</td>
<td>5627KB</td>
<td>5628KB</td>
</tr>
<tr>
<td>Number of Procedures</td>
<td>12455</td>
<td>12891</td>
<td>12891</td>
<td>12902</td>
<td>12965</td>
<td>12966</td>
</tr>
<tr>
<td>Number of Code Blocks</td>
<td>145111</td>
<td>155996</td>
<td>156044</td>
<td>156175</td>
<td>157417</td>
<td>157417</td>
</tr>
<tr>
<td>Number of Data Blocks</td>
<td>9357</td>
<td>9440</td>
<td>9440</td>
<td>9445</td>
<td>9511</td>
<td>9512</td>
</tr>
<tr>
<td>Maximum Procedure</td>
<td>96.06%</td>
<td>99.42%</td>
<td>99.42%</td>
<td>99.51%</td>
<td>99.99%</td>
<td>--</td>
</tr>
<tr>
<td>Maximum Code</td>
<td>92.18%</td>
<td>99.10%</td>
<td>99.13%</td>
<td>99.21%</td>
<td>100.00%</td>
<td>--</td>
</tr>
<tr>
<td>Maximum Data</td>
<td>98.37%</td>
<td>99.24%</td>
<td>99.24%</td>
<td>99.30%</td>
<td>99.99%</td>
<td>--</td>
</tr>
<tr>
<td>Matching Procedure</td>
<td>92.21%</td>
<td>99.02%</td>
<td>99.02%</td>
<td>99.11%</td>
<td>99.99%</td>
<td>--</td>
</tr>
<tr>
<td>Matching Code</td>
<td>93.01%</td>
<td>99.51%</td>
<td>99.50%</td>
<td>99.55%</td>
<td>100.00%</td>
<td>--</td>
</tr>
<tr>
<td>Matching Data</td>
<td>90.52%</td>
<td>97.90%</td>
<td>97.90%</td>
<td>97.95%</td>
<td>99.97%</td>
<td>--</td>
</tr>
</tbody>
</table>

Table 4.3. Matching rates for six versions of mshtml.dll. In this experiment, we ran BMAT between the latest build (08/11/99) and each previous version.

Maximum Remain: Total number of units (procedures, code blocks or data blocks) in the older build divided by total number in the newer build. This ratio represents the maximum percentage of units in the newer build that have remained the same since the older build. The percentage of units that are actually the same is most likely lower, due to units that have been deleted or changed.

Matching Rate: Percentage of units in the newer build for which we successfully find matches, regardless of whether the matches turn out to be correct. For procedures and data blocks, the Maximum Remain gives an absolute upper limit for the Matching Rate, since the mapping is one-to-one. For code blocks, the Matching Rate may exceed the Maximum Remain because multiple blocks may be matched to the same block during control flow-based basic block matching.

For two cases of binary matching, Table 4.4 and Table 4.5 list the matching rates on the procedure and code block levels, respectively, after different phases of the matching process. These numbers demonstrate the performance gain of BMAT from using a multiple-pass approach and a combination of different methods. All results are reported in terms of matching rates in the newer build, because we are usually interested in propagating profile information from the older build to the newer build. A higher matching rate in the newer build enables us to propagate more information.
Table 4.4. Procedure matching rates at different stages. This table lists the number and percentage of successfully matched procedures in the newer build after using each of the four methods. The four methods are used sequentially from left to right, and each number reflects the cumulative result of all methods that have been used up to that point. The results show that most matches can be found by simply looking for identical names, while the contents-based method noticeably improves the results. As mentioned in Section 4.1, block trial-matching, which can be time-consuming, often finds very few additional matches.

In the case of shell32.dll, our investigation shows that a large number of procedures were removed from or added to the program between the two builds. These procedures cannot be matched by our algorithm.

Table 4.5. Code block matching rates at different stages. This table lists the total number and percentage of successfully matched code blocks in the newer build for three cases: after using only the first hashing pass (fuzziness level 1), after using all hashing passes, and after using both hashing and control flow-based matching. These results demonstrate benefits from using multiple matching passes and performing control flow-based basic block matching.

In the case of mshtml.dll, the number of matched code blocks in the newer build after control flow-based matching exceeds the total number of code blocks in the older build. This is a result of multiple blocks being matched to the same block. Most code blocks that remain unmatched after control flow-based matching are blocks in unmatched procedures.
4.3.3. Profile Propagation

We ran an automated test on Internet Explorer 5.0 using five different versions of mshtml.dll, and collected profiles for each version. Between the latest build and each earlier build, we used BMAT to propagate the profile for the older version onto the newer version, then compared the propagated profile with the profile collected directly on the newer version. We use two of the metrics we introduced in Chapter 3. The first one is an optimization-specific metric that measures the effectiveness of the profile in guiding static branch prediction. We perform static branch prediction on the new version using both the propagated profile and the directly collected profile, and calculate the dynamic success rate of each prediction. The Branch Prediction metric ($B.P.$) is the ratio of the two success rates, i.e., the success rate using the propagated profile divided by the success rate using the directly collected profile. This metric measures how well BMAT works on the most frequently executed part of the program. The second one is the Correct Coverage Percentage ($C.C.$) that measures the percentage of basic blocks for which the propagated profile correctly indicates if it has a non-zero execution count, i.e., if it gets executed under the given workload. This metric measures how well BMAT works across the whole binary.

On an interactive application like Internet Explorer, even an automated test may generate slightly different profiles from run to run. We call this the execution uncertainty factor. To quantify this effect, we ran the test three times on each version, and compared the three profiles to each other using the metrics defined above. The calculation of the metrics remains the same, with one profile collected earlier assumed to be “propagated profile” and one collected later assumed to be “directly collected profile.”

Results from both profile propagation and self-comparison are given in Table 4.6 (see next page). In these experiments, error rate from profile propagation is under 0.13% for branch prediction, about 16 times the largest execution uncertainty factor we see when the test is repeated on the same version. For code coverage, the error rate is under 0.5%, less than three times the largest execution uncertainty factor. If these error rates are acceptable for the purpose of optimization, all profile information collected for mshtml.dll during the five-month period can be propagated and used on the latest build.

Table 4.7 (see next page) presents the matching rates and profile propagation results for eight DLLs from Microsoft Windows 2000. Even though the oldest build A and the newest build D are only six weeks apart, the average matching rate across the eight DLLs (98.92%) is lower than the matching rate for two builds of mshtml.dll that are five months apart (99.51%, Table 4.3). This suggests that this group of DLLs underwent more significant changes during the six weeks than mshtml.dll did during the five months. Consequently, the profile propagation results are not as close to 100% as the results for mshtml.dll, but they are still very promising.

---

8 All experiments were run on the same machine, using the same version of Internet Explorer with different versions of mshtml.dll. The 10/22/98 version was excluded because of a compatibility problem. The automated test exercises about 27% of the code blocks in mshtml.dll.

9 If a branch in the new version is not covered by the propagated profile, we predict the branch to be “taken.”

10 If a basic block in the new version is not covered by the propagated profile, we assume that it has a zero execution count.
Table 4.6. Profile propagation and self-comparison for five versions of mshtml.dll.

**B.P.**: Static branch prediction success rate using propagated profile divided by the rate using directly collected profile.

**C.C.**: Accuracy of code coverage data from the propagated profile as compared to the directly collected profile. Each code block in the new version is “agreed” if its coverage bit is the same according to the propagated profile or the directly collected profile. **C.C.** is the percentage of “agreed” code blocks among all code blocks in the new version.

**Matched against self**: Comparison results between different test runs on the same build. We compare profiles from two runs using the above metrics. Each number listed is the average of three pair-wise comparisons between three runs. These results quantify the *execution uncertainty factor*.

<table>
<thead>
<tr>
<th>Build Date</th>
<th>03/15/99</th>
<th>04/16/99</th>
<th>07/15/99</th>
<th>07/27/99</th>
<th>08/11/99</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matched</td>
<td><strong>B.P.</strong></td>
<td>99.997%</td>
<td>99.997%</td>
<td>99.997%</td>
<td>99.992%</td>
</tr>
<tr>
<td></td>
<td><strong>C.C.</strong></td>
<td>99.94%</td>
<td>99.95%</td>
<td>99.96%</td>
<td>99.83%</td>
</tr>
<tr>
<td>Matched</td>
<td><strong>B.P.</strong></td>
<td>99.876%</td>
<td>99.876%</td>
<td>99.894%</td>
<td>99.998%</td>
</tr>
<tr>
<td>against 08/11/99</td>
<td><strong>C.C.</strong></td>
<td>99.59%</td>
<td>99.61%</td>
<td>99.55%</td>
<td>99.93%</td>
</tr>
</tbody>
</table>

Table 4.7. Matching rates and profile propagation for eight Windows 2000 DLLs.

A, B, C, D: Four different builds, in time order, over six weeks of time (05/11/99, 06/04/99, 06/15/99, 06/22/99)

**M.R.**: Matching Rate for code blocks, percentage of successfully matched code blocks in the new build (D)

**B.P.**, **C.C.**: Branch Prediction and Correct Coverage metrics for profile propagation (see Table 4.6 caption)
Throughout Table 4.7, most branch prediction metrics are over 99% and most code coverage metrics are over 98%. Between builds A and D which are six weeks apart, five out of the eight DLLs produce branch prediction and code coverage metrics both over 97%. For the other three, one of the two metrics is over 97%. On average, between builds A and D, the propagated profiles are 98.8% up to par in terms of static branch prediction and 97.7% accurate in terms of code coverage information. Unsurprisingly, the averaged metrics are even higher for profile propagation from a more recent build (B or C).

**Summary**

This chapter presented BMAT, a code contents-based binary matching tool, and experiments on profile propagation using BMAT. Our results showed that using binary matching and profile propagation, it is often possible to obtain highly accurate profile information based on profiles for previous versions that are weeks or months old. Such high accuracy is achieved by combining several methods of matching and using multiple hashing passes. When the old and new builds are further apart in time, the accuracy of matching and the quality of propagated profile tend to decrease. The results also suggested that the rate of program changes, and consequently the potential for profile propagation, varies from case to case.
Chapter 5. Control Flow-based Matching Algorithm

BMAT for the most part matches two program versions based on their code contents, regardless of any changes in the program’s control flow structure. As discussed in Section 2.4, another possible approach to binary matching is to compare the control flow structure of the two versions. This approach can capture changes to a program’s control flow and thus allow an accurate understanding of how the code behavior will change because of program revision.

This chapter presents a binary matching algorithm that is mostly based on the control flow structure of each procedure, while still using code contents to help find correct matches. Due to both practical reasons and our desire to explore progressive profiling on various platforms, we implemented this algorithm on the DEC Alpha architecture instead of the x86 architecture. We use the ATOM tool [Srivastava and Eustace 1994] only to help dissecting Alpha binaries. The binary rewriting function of ATOM is not used.

In designing this algorithm we skip data matching and focus on code matching only. We also choose not to implement sophisticated methods of procedure matching because manual examination of our benchmark programs reveals few cases of procedure name changes. We perform procedure matching based on their names only, and use the spatial order of the procedures to capture a few cases of procedure name changes. We then conduct intra-procedural code matching within each pair of matched procedures.

Next we define the units of our control flow-based code matching, then discuss how we conduct the matching for each procedure. We then present our experimental results on profile propagation using the control flow-based binary matching tool.

5.1. Defining Code Blocks (Units of Matching)

Changes in program control flow structure often involve moving a chunk of code either into or out of a control flow region, which can be an if statement or a loop. Our definition of code blocks, i.e. the smallest code units for matching, should be based on the need to identify such changes.

Our code blocks are based on program basic blocks, but with two major adjustments. During program execution, most subroutine calls return successfully and do not affect the control flow within a procedure. In addition, subroutine calls are often deleted from, added to or moved within a procedure, and sometimes replaced with other codes due to procedure inlining. These changes may confuse the matching algorithm if subroutine calls are used to identify code blocks. Therefore, our code block combines multiple basic blocks that are separated only by subroutine calls.

Another adjustment is based on the fact that in terms of control flow region, a control flow instruction at the end of a basic block and the rest of the basic block are not necessarily related to each other. For example, the control flow instruction may be the condition clause of an

---

11 Assuming that programmers do not move procedures around in the source code, we can expect the sequence of procedures in the binary code to mostly remain the same. If we see procedure sequence “foo”, “cat”, “bar” in the old version and “foo”, “dog”, “bar” in the new version, and neither “cat” nor “dog” is matched to another procedure, it is likely that “cat” has been renamed to “dog”.
if statement, while the rest of the basic block (or part of it) comes from preceding program code that has nothing to do with the if statement. As another example, a control flow instruction may be added to or removed from the end of a basic block, while the rest of the basic block remains the same. For this reason, we separate the control flow instruction from the rest of the basic block and view it as part of the procedure’s control flow structure. When we talk about code block contents in this chapter, we implicitly exclude the control flow instructions at the end.

Based on the above criteria, we use the following process to divide a procedure into code blocks. We start with basic blocks as they are divided without using subroutine calls as separators. If a basic block ends with an unconditional branch, we simply exclude that instruction from the code block and annotate the block with the branch target. This information will be used to analyze the procedure’s control flow structure. If a basic block ends with a conditional branch, we separate the branch instruction into a special shadow block called condition block, which may become the condition clause of an if statement or a loop. The rest of the basic block becomes one code block. If the conditional branch instruction is immediately preceded by a compare instruction that sets the source register of the branch instruction, we put the compare and branch instructions both in the condition block, as they are tightly associated with each other and are most likely compiled from the same statement in the source code.

Note that as we exclude the control flow instructions, a code block may become empty. We keep these empty code blocks because they correspond to non-empty basic blocks that may carry profile information.

Figure 5.1 (see next page) shows two versions of a procedure, both broken down to code blocks. Later in this chapter we will use this example again to demonstrate our matching algorithm.
Figure 5.1. Example of code blocks.

Unconditional branches are listed here for reference, but they are not considered as part of the code blocks.
5.2. Constructing Control Trees

A procedure’s control flow structure is commonly expressed as a Control Flow Graph (CFG), a directed graph with code blocks as nodes and control flow transfers as edges. Using this method, the problem of procedure matching becomes a problem of pattern matching between two graphs. However, general graph matching is a NP-complete problem. Furthermore, the spatial order of the code blocks, which is an important clue in binary matching, is lost in the CFG. One possible solution is to add the spatial information into the graph as special edges. However, this will complicate the graph matching problem even further.

In this project, we use control tree [Muchnick 1997] to express the control flow structure of a procedure. Tree matching is less complex than graph matching. Even though some unordered tree matching problems are still NP-complete, ordered tree matching can generally be solved in polynomial time [Kilpelainen 1992]. Note that control tree representation does not use nodes and edges to represent the same things as they do in CFG. In a control tree, leaf nodes are program code blocks while non-leaf nodes are control flow regions such as if statements and loops. The tree edges represent the nesting relationship between control flow regions and code blocks. In other words, the children nodes of a control flow region are all the control flow sub-regions and code blocks directly contained in that region. The specifics about control flow transfers are preserved in the non-leaf nodes. In addition, a control tree also indicates the spatial order of the code blocks through the order of sibling nodes.

To build our control tree, we use a limited form of structural analysis [Muchnick 1997] that only recognizes various forms of if statements and natural loops. If a procedure has no control flow transfers other than these two types, all control flow regions in the procedure form a nesting hierarchy. Our structural analysis can recognize such a hierarchy and convert it to a control tree. On the other hand, statements such as break, continue, return and goto are compiled into unconditional branches that we call irregular jumps. Some irregular jumps may lead to irreducible structures or arbitrary acyclic structures [Muchnick 1997], which our structural analysis does not recognize. Figure 5.2 (see next page) illustrates this case.

In our analysis, if we encounter an unconditional branch that leads to unrecognizable control flow structures, we conclude that it is an irregular jump. We then exclude this unconditional branch from our structural analysis, and add an annotation to the code block that includes the irregular jump. This annotation will later be used by the block matching algorithm. This approach is acceptable because we do not have to accurately identify all control flow structures. Our ultimate goal is profile propagation, and thus our analysis need only lead to correct matches for the code blocks.

Irregular jumps do not always lead to unrecognizable control flow structures. Sometimes they make a group of code blocks look like an if statement or a loop even though it was not in the source code. Based on the binary code alone, it is often impossible to make the distinction. In our algorithm, we use some heuristics to detect irregular jumps in an effort to simplify the control tree. For example, multiple backward-goto instructions with the same target may appear like a deep-nesting loop hierarchy with all nested loops starting at the same block. Since such structures are rare in real programs, we treat this case as multiple gotos instead of a loop hierarchy, which makes the control tree simpler and the matching process less prone to errors.
Figure 5.2. Unrecognizable control flow structures due to irregular jumps. In the control flow graph on the left, blocks \([a2_{\text{cond}} : a6_{\text{cond}}]\) constitute a loop. Block \(a2_{\text{cond}}\) is compiled from a \textit{break} within an \textit{if} statement. Block \(a3_{\text{cond}}\) is from a \textit{goto} within an \textit{if} statement. Block \(a5\) ends with a \textit{return} statement, which is compiled into a jump to the last block in the procedure. In the graph on the right, blocks \([a1_{\text{cond}} : a6_{\text{cond}}]\) constitute a loop. Block \(a3\) ends with a \textit{continue} statement, which causes the control flow to jump outside the nested \textit{if} hierarchy. In both examples, irregular jumps break the nesting hierarchy of control flow regions, and our structural analysis cannot directly build a control tree. One possible solution is to duplicate code blocks and create recognizable control flow regions. We choose not to use this method because such duplication will complicate the block matching process.

Switch statements are transformed by the compiler into complex \textit{if} structures with some irregular jumps. Therefore, our algorithm is capable of dealing with these cases. Function pointers translate to jump instructions using register addressing. Since we are unable to detect their targets in our static analysis, we include these jumps in annotations but not the control tree.

Loop structures can take several different forms in the binary code. In the source code there can be \textit{for} / \textit{while} loops, \textit{do-while} loops, and infinite loops caused by a backward \textit{goto}. In addition, the compiler has several different ways of compiling a loop. Based on our observation, we summarize four different forms of loops in binary code, as illustrated in Figure 5.3 (see next page). Once we recognize each of two loop structures as one of the four forms, we can match them accordingly.

As an example, Figure 5.4 shows the control trees constructed from the sample code we used in Figure 5.1.
Figure 5.3. Four different forms of loop. B1 and B2 are equivalent structures.

A: do-while loop
B1: for/while loop
B2: for/while loop with condition clause moved below the loop body
C: for/while loop with condition clause moved down and peeled once
D: infinite loop with an unspecified number of exit edges in the loop body

All four loop forms can be matched to each other with minor adjustments.

The initialization clause of a for loop may not be included in the structure because it is difficult to distinguish from an unrelated statement preceding the loop.

An if statement that encapsulates a do-while loop may be regarded as a skip block in loop form C, as the two cases are sometimes indistinguishable. In this case, however, our matching algorithm will still be able to find the correct matches.
Figure 5.4. Control trees for the code in Figure 5.1.
In the old version, \([a_2 \text{cond}: a_3 \text{cond}]\) is recognized as an instance of loop form C.
In the new version, \([b_4:b_5]\) is recognized as an instance of loop form B1.
5.3. Matching Control Trees

After constructing control trees, the problem of finding a mapping between two versions of a procedure translates to pattern matching between two trees. The fact that we take code contents into consideration means that the tree nodes, i.e. the code blocks, are labeled. Since we want to find not just exact matches but also fuzzy matches, our problem is similar to approximate tree pattern matching [Shasha and Zhang 1997].

The hierarchical structure of a control tree represents the nesting structure of control flow regions in the code, which should be preserved in our profile propagation. Therefore, our matching results should preserve the parental order of the tree nodes. In other words, if node $a$ in one tree is matched to node $b$ in another tree, any match for an offspring of $a$ should be an offspring of $b$. There are exceptions to this rule, however, as we will discuss later in this section.

The goal of preserving parental order is common among most tree matching problems. However, there are several important differences between our control tree matching problem and a general tree matching problem, which make it problematic to use existing tree matching algorithms for our purpose.

In a general tree matching problem, each node carries a symbolic label, and the nodes can be directly compared to each other for a result of equal or not equal. In our control tree matching, the tree nodes represent binary code blocks, which often change from version to version. It is not sufficient to know if two code blocks are strictly the same; we also need to look at how different they are. It is difficult to quantify such difference using just one criterion, as there are numerous ways in which two code blocks can differ from each other.

Secondly, typical algorithms for tree matching only consider one-to-one mapping. In our binary matching, the correct matches between the code blocks may not be one-to-one, especially when we look for fuzzy matches.

A third issue is that general tree matching problems are divided into ordered and unordered tree matching depending on whether the order of sibling nodes matters to the matching. The two categories are solved with different algorithms. Our control tree matching problem does not strictly fit into either one. In a control tree, the children nodes of a control flow region are code blocks and control flow sub-regions within that region. They are usually in equivalent positions in terms of control flow, and therefore their relative order does not matter to our binary matching. In some cases, however, the order of code blocks does matter, such as the then and else clauses of an if statement or code blocks separated by an irregular jump. In addition, there are often data dependencies between code blocks that restrict their relative order. If we consider the control tree matching problem as unordered tree matching, the problem becomes NP-complete, and we run into an increased number of ambiguous cases. If we consider it as ordered tree matching, we miss the opportunity of matching two control trees that are equivalent except for the order of certain sibling nodes.

These three differences suggest that control tree matching is more complicated than a typical tree matching problem. In order to design a proper algorithm, we first make some observations on comparing two versions of a procedure, assuming there are no radical changes between the two versions.

If there are some changes to code block contents but no changes to the procedure’s control flow structure, the execution pattern will remain the same. Our algorithm should be able
to find a complete, one-to-one match based on the fact that the two control trees have the same structure. On the same token, if a pair of sub-trees within the two control trees have the same structure, they likely represent matching code regions. However, one procedure may contain several code regions with identical control flow structure. This may cause confusion, especially if we consider the possibility that two code regions within one procedure may switch places.

If there are changes to a procedure’s control flow structure, it will affect the procedure execution pattern, even though the code contents may remain the same (except for instructions directly involved in the control flow changes). In many cases, one change in control flow structure causes a region of code to shift uniformly from one execution path to another. In the control tree, this looks like a transplanting process where a sub-tree or sub-forest is moved to a different location. Transplanting of a single node is relatively easy to deal with, as we can deduce its new execution pattern based on its new environment. Transplanting of a non-trivial sub-tree, however, is an interesting case. If the control flow structure within a transplanted sub-tree does not change, we speculate that it maintains the same execution pattern, in which case the execution counts for code blocks within the sub-tree will scale proportionally in the new version. By detecting a transplanted sub-tree, we can generate a conversion formula for code blocks in that sub-tree. However, the detection of transplanted sub-trees is complicated by the fact that the structure of the sub-tree itself may also change. Figure 5.5 shows an example of a transplanted sub-tree and the associated profile conversion formula.

![Figure 5.5. Profile conversion formula for a transplanted sub-tree.](image)

In reality, there are often changes in both code block contents and control flow structure. Sometimes there are many possible ways of matching for one code region. Many modern application programs have a lot of small code blocks that are similar to each other. This further increases the likelihood of facing multiple matching choices, and consequently the danger of finding wrong matches.
Based on the above observations, our algorithm must be flexible enough to find approximate matches between sub-trees, but also be able to distinguish multiple sub-trees that appear to be similar. Most of our matches need to preserve the parental order of tree nodes, but it is important to detect the cases of transplanted sub-trees.

One of the most basic solutions to tree pattern matching is the bottom-up method, which starts by finding matching nodes at the leaf level and then traverses up the tree to find larger and larger matching patterns. To increase our opportunities for finding fuzzy matches, we expand the bottom-up method to a two-phase algorithm. First we use a bottom-up phase to find one or more matching candidates for each node in the control tree for the new version, then we use a top-down phase to finalize the matches. When we look for matching candidates in the bottom-up phase, multiple selections for lower-level nodes increase the chance of finding fuzzy matches at higher levels. When we finalize all matches in the top-down phase, the higher-level matches serve as a guide to selecting matches for lower-level nodes.

Next we discusses the two phases in more detail. Some heuristic decisions in the algorithm design are based on architecture characteristics, compiler conventions and certain assumptions on programmers’ behaviors.

For simplicity, we use new control tree and old control tree to refer to the control tree for the new version and the old version of a procedure, respectively.

**Bottom-Up Phase: Finding Matching Candidates**

In the bottom-up phase, we take each node in the new control tree and look for one or more matching candidates for that node. In order to find the best candidates, we pair the node up with each node in the old control tree and calculate several similarity measurements for them. We then choose one or more nodes that produce the highest similarity measurements.

In the original bottom-up method, the match for each node \( x \) is based on the entire sub-forest that is rooted from all children nodes of \( x \). We call this the children sub-forest of \( x \). For the purpose of flexibility, we reduce the matching criteria to selected statistics of \( x \)’s children sub-forest plus the matching status of \( x \)’s children nodes. This accommodates minor changes in either tree structure or node contents so that we can find approximate matches, and still allows us to use the hierarchical structure of the tree to identify correct matches. Since we do not need all the details of the children sub-forest, we eliminate a major drawback of the original bottom-up method, i.e., the fast growth of table spaces.

For each control tree node \( x \), we calculate a series of statistics from its children sub-forest. Five of them are based on the structure of the sub-forest, independent of the contents of the code blocks:

- \( \text{Number}_{-}\text{Subif}(x) \) = number of if nodes among the children of \( x \)
- \( \text{Number}_{-}\text{Subloop}(x) \) = number of loop nodes among the children of \( x \)
- \( \text{Depth}_{-}\text{Subif}(x) \) = maximum depth of nested if structures
- \( \text{Depth}_{-}\text{Subloop}(x) \) = maximum depth of nested loop structures
- \( \text{Number}_{-}\text{Blocks}(x) \) = total number of code blocks covered by the sub-forest
These five statistics are selected so that for most types of program changes, a single change will affect only one or two among the five. We also use three statistics that are based on the code contents:

- **Number_Instrs (x)** = total number of instructions covered by the sub-forest
- **Total_Weight (x)** = total “weight” of all instructions in the sub-forest

(We assign a “weight” to each type of instruction based on how likely it is to characterize the code block it is in. The more likely a type of instruction may change between program versions, the less weight it contributes. Based on our observations, we assign highest weights to return instructions and subroutine calls, medium weights to loads and stores, and lowest weights to arithmetic instructions.)

- **Total_Immed (x)** = sum of all immediate operands in non-memory instructions

(This statistic distinguishes code blocks that are identical except for immediate operands. We exclude immediate operands in load and store instructions, as they often change from version to version due to the shifting of code and data in the address space.)

None of the eight statistics defined above is dependent on the order of x’s children nodes. In other words, the bottom-up phase views the control tree as a non-ordered tree. To compare the statistics from different control tree nodes, we introduce a *Min-to-Max* function similar to the *Max-to-Min* function we defined in Chapter 3:

\[
\text{Min-to-Max} (p, q, C) = \begin{cases} 
\frac{(p + C)}{(q + C)}, & \text{if } p < q \\
\frac{(q + C)}{(p + C)}, & \text{if } p \geq q
\end{cases}
\]

where \( C > 0 \) is a constant that determines the sensitivity of the function. It is obvious that \( 0.0 < \text{Min-to-Max} (p, q, C) \leq 1.0 \).

Given a pair of control tree nodes \( a \) and \( b \), we calculate a structure-based similarity measure **SM_Structure** and a contents-based similarity measure **SM_Contents**:

\[
\text{SM_Structure}(a, b) = \text{Min-to-Max} (\text{Number_Subif}(a), \text{Number_Subif}(b), C1) \\
* \text{Min-to-Max} (\text{Number_Subloop}(a), \text{Number_Subloop}(b), C2) \\
* \text{Min-to-Max} (\text{Depth_Subif}(a), \text{Depth_Subif}(b), C3) \\
* \text{Min-to-Max} (\text{Depth_Subloop}(a), \text{Depth_Subloop}(b), C4) \\
* \text{Min-to-Max} (\text{Number_Blocks}(a), \text{Number_Blocks}(b), C5)
\]

\[
\text{SM_Contents}(a, b) = \text{Min-to-Max} (\text{Number_Instrs}(a), \text{Number_Instrs}(b), C6) \\
* \text{Min-to-Max} (\text{Total_Weight}(a), \text{Total_Weight}(b), C7) \\
* \text{Min-to-Max} (\text{Total_Immed}(a), \text{Total_Immed}(b), C8)
\]

Based on the definitions we have

\( 0.0 < \text{SM_Structure}(a, b) \leq 1.0, \quad 0.0 < \text{SM_Contents}(a, b) \leq 1.0 \)

The higher these two measurements, the more similarity there is between \( a \) and \( b \). Measurements of 1.0 indicate that all statistics are equal for the two nodes. The constants \( C1, C2, \ldots, C8 \) should reflect the assumed likelihood of various types of program changes. We set them based on observations and experiments on our test programs.
In addition to these two similarity measurements, we also use the matching status of children nodes to help evaluate potential matches. If we find a match between a pair of nodes, it is likely that their parent nodes are good candidates for matching. Therefore, we define a third similarity measure $SM_{Children}$. For a pair of non-leaf nodes $a$ and $b$, $SM_{Children}(a, b)$ is the average of $SM_{Composite}(aa, bb)$ for all $aa$ and $bb$ where $aa$ is a child node of $a$, $bb$ is a child node of $b$, and $aa$-$bb$ is already selected as a possible match. $SM_{Composite}$ is an “overall” similarity measurement defined as

$$SM_{Composite}(a, b) = SM_{Structure}(a, b) * SM_{Contents}(a, b) + 0.1 * \text{Truncate}(SM_{Children}(a, b))$$

The constant that we set as 0.1 determines how much the children nodes weigh in on our decision. Truncate( ) is a function that converts any value larger than 1.0 to 1.0, so that $SM_{Composite}$ never exceeds an upper limit. In this case the limit is 1.1.

$SM_{Composite}$ is the main similarity measurement we use to choose matching candidates for each node. However, we also use $SM_{Structure}$ as a preliminary filter to guarantee that the matching candidates are similar in their control flow structure. Specifically, for each node in the new control tree, we identify a number of nodes in the old control tree that yield relatively high $SM_{Structure}$ measurements. Among these preliminary candidates, we use the $SM_{Composite}$ measurement to determine which one(s) among them are worthy candidates for matching.

It is not uncommon that a procedure includes several code blocks with the same code contents. In this case, each of these code blocks in the old version will match with each one in the new version, forming a bipartite matching pattern between the two groups. In an effort to save storage space for matching candidates, we look for any bipartite matching patterns after we finish each tree level, and break each pattern into one-to-one matches based on the order of the code blocks.

**Top-Down Phase: Finalizing the Matches**

In the top-down phase, we traverse the new control tree in depth-first order and finalize the block matches recursively. This gives us the opportunity to use higher-level matches to guide the final selection of lower-level matches.

Even though the selection of final matches is mainly based on the similarity measurements calculated in the bottom-up phase, we also try to preserve the parental order of tree nodes as we discussed earlier in this section. If node $a$ in the old control tree and node $b$ in the new control tree are already selected as a match and $bb$ is a child node of $b$, we give priority to the children nodes of $a$ when selecting a final match for $bb$. If we fail to find a good match among them, $bb$ may be the root of a transplanted sub-tree. We then look for matches among the grandchildren and higher-generation offspring of $a$, and finally the rest of the old control tree.

We also use the sibling order of tree nodes to help selecting matches. For most procedures, the spatial order of its code blocks remains mostly the same from version to version. Therefore, if we fail to find matches for a tree node, we use already-selected matches of its sibling nodes to find more matching candidates. This process is similar to the propagation phase in our code content-based binary matching as discussed in Section 4.2.3. This also means that unlike the bottom-up phase, the top-down phase views the control tree as an ordered tree.
Figure 5.6 displays the pseudo-code for our two-phase control tree matching algorithm, while Figure 5.7 uses an example to illustrate the process.

```
// tree_old, tree_new : control trees for the old version and the new version
// procedure_old, procedure_new : level 1 (root) nodes of tree_old and tree_new

control_flow_tree_matching (tree_old, tree_new)
{
    // bottom-up phase
    matching candidate set MC = {};
    for (int i = depth of tree_new; i >= 2; i --) {
        for each level i node n_new in tree_new {
            find one or more possible matches among all nodes in tree_old;
            for each possible match n_old, enter (n_old, n_new) into MC;
        }
        if there are "bipartite group match" patterns in MC {
            try to break down each group match into one-to-one matches;
        }
    }

    // top-down phase in a recursive fashion
    children_matching (procedure_old, procedure_new);
}

// recursive matching subroutine for top-down phase
children_matching (parent_old, parent_new) {
    for each node child_new that is a child node of parent_new {
        choose child_new’s final match n_old; // priority given to children of parent_old
        if n_old is not a child of parent_old {
            annotate child_new as “transplanted” and create profile conversion formula;
        }
    }

    assign matches to any unmatched child node of parent_new; // “match propagation”
    for each child_new that is a non-leaf child node of parent_new {
        children_matching (final match for child_new, child_new)
    }
}
```

Figure 5.6. Pseudo-code for the two-phase control tree matching algorithm.
The bottom-up phase looks for matching candidates:

<table>
<thead>
<tr>
<th>Level</th>
<th>Added Matching Candidates</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>a1–b4, a3–b5</td>
</tr>
<tr>
<td>3</td>
<td>a2–b2, a2–b3, loop_a–loop_b, a4–b6</td>
</tr>
<tr>
<td>2</td>
<td>a1–b1, if_a–if_b, a5–b7</td>
</tr>
<tr>
<td>1</td>
<td>--</td>
</tr>
</tbody>
</table>

The top-down phase finalizes all matches (F denotes a profile propagation formula):

<table>
<thead>
<tr>
<th>Level</th>
<th>Added Matches</th>
<th>Call to children_matching( )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>--</td>
<td>procedure_a, procedure_b</td>
</tr>
<tr>
<td>2</td>
<td>a1–b1, if_a–if_b, a5–b7</td>
<td>if_a, if_b</td>
</tr>
<tr>
<td>3</td>
<td>F(a1,a2)–b2, a2–b3, loop_a–loop_b, a4–b6</td>
<td>loop_a, loop_b</td>
</tr>
<tr>
<td>4</td>
<td>F(a2,a3)–b4, a3–b5</td>
<td>--</td>
</tr>
</tbody>
</table>

Block b2 has no direct match, but we can generate a formula based on the if structure:

\[
\text{count}(b2) = \text{count}(a1) - \text{count}(a2)
\]

For block b4, the formula is derived from the different forms of the two loop structures:

\[
\text{count}(b4) = \text{count}(a2) + \text{count}(a3)
\]

**Figure 5.7. An example of control tree matching.** The original code for the two versions can be found in Figure 5.1. Since condition blocks are shadow blocks attached to other code blocks, we do not look for matches for them. However, their code contents are taken into account when we match the if and loop nodes.
5.4. Experimental Results

One benchmark we use is cc1, a program module from the GNU C/C++ compiler suite gcc. Cc1 is a front-end module that converts C program to assembly code. For our experiments we use three versions of cc1 module, taken from versions 2.8.0, 2.8.1, and 2.95 of the gcc suite. The version numbers suggest that there are probably only minor changes between versions 2.8.0 and 2.8.1 but significant changes between versions 2.8.1 and 2.95. We also use two versions of the GNU ghostscript program as another benchmark case. All source code was downloaded from GNU’s ftp site. All versions were compiled on the same Alpha EV56 machine running DEC OSF 4.0 operating system, using the provided instructions and default settings. The table below summarizes the five different program versions as our benchmark.

<table>
<thead>
<tr>
<th>Benchmark Program</th>
<th>cc1</th>
<th></th>
<th>ghostscript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>2.8.0</td>
<td>2.8.1</td>
<td>2.95</td>
</tr>
<tr>
<td>Release Date</td>
<td>01/08/1998</td>
<td>03/03/1998</td>
<td>07/29/1999</td>
</tr>
<tr>
<td>File Size</td>
<td>3104KB</td>
<td>3104KB</td>
<td>7568KB</td>
</tr>
<tr>
<td>Code Segment Size</td>
<td>1512KB</td>
<td>1496KB</td>
<td>2008KB</td>
</tr>
<tr>
<td>Number of Procedures</td>
<td>2314</td>
<td>2318</td>
<td>2874</td>
</tr>
<tr>
<td>Number of Code Blocks</td>
<td>56589</td>
<td>56358</td>
<td>65939</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12/05/1997</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>12/17/1998</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2288KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2576KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1304KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1392KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3759</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4269</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>39650</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>41915</td>
</tr>
</tbody>
</table>

Table 5.1. Summary of benchmark programs. Release Date is the latest date stamp in the source file tree. As discussed in Section 5.1, our code blocks are essentially program basic blocks with subroutine calls not considered as basic block separators.

Based on the principle of training and testing on different workload, we did not tune the heuristics and parameters in our algorithm specifically for the benchmark programs. Instead we used the email program pine, program module cpp from the gcc suite, and a few other small programs.

5.4.1. Running Time

Table 5.2 (see next page) lists the running time of the matching algorithm for our three test cases. All experiments were run on an Alpha EV56 machine running DEC OSF 4.0 operating system. For all test cases, the total running time is under half a minute.
Table 5.2. Algorithm running time. All running times are measured using the *time* system utility in DEC OSF, and are average results from five test runs. All standard deviations are under 0.1 second.

<table>
<thead>
<tr>
<th>Program Versions</th>
<th>cc1 2.8.0</th>
<th>cc1 2.8.1</th>
<th>cc1 2.8.1</th>
<th>cc1 2.95</th>
<th>ghostscript 5.0</th>
<th>ghostscript 5.10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Segment Size</td>
<td>1512KB</td>
<td>1496KB</td>
<td>1496KB</td>
<td>2008KB</td>
<td>1304KB</td>
<td>1392KB</td>
</tr>
<tr>
<td>Running Time (Second)</td>
<td>Build 4.7</td>
<td>Build 4.7</td>
<td>Build 4.7</td>
<td>Build 5.6</td>
<td>Build 6.5</td>
<td>Build 7.1</td>
</tr>
<tr>
<td></td>
<td>Match 10.4</td>
<td>Match 10.6</td>
<td>Match 10.6</td>
<td>Match 6.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total 19.8</td>
<td>Total 20.9</td>
<td>Total 20.9</td>
<td>Total 20.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Speed (Second/MB)</td>
<td>13.5</td>
<td>12.2</td>
<td>15.5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.4.2. Procedure Matching Results

As discussed in Section 2.3, the one-to-one procedure mapping produced by procedure-level matching may not cover either version of the program completely. Not included are procedures that are deleted from or added to the program, plus procedures that have undergone name changes and not been recognized.

Among the procedures included in the mapping, some have exactly the same control flow structure in both versions. We use the term *consistent procedures* to describe them. Even though there may be code changes in these procedures, the correct way of code matching is simply to match the code blocks one by one in the order they appear. In other words, the success rate of code matching within a consistent procedure is always 100%.

In our report on code block matching results in the next section, we will exclude consistent procedures and report only procedures with changes in control flow structure, which we call *restructured procedures*.

Table 5.3 summarizes the procedure matching results for our test cases. Unsurprisingly, larger procedures are more likely to undergo changes in control flow structure. We also see that the difference between cc1 2.8.1 and 2.95 is much larger than that between cc1 2.8.0 and 2.8.1, demonstrated by a larger portion of procedures missing from the one-to-one mapping and a larger portion of restructured procedures.

---

12 Note that “success rate” refers to the percentage of code blocks for which we find a likely match, which may or may not be the correct match in the end.
### Table 5.3. Procedure matching results

Only procedures in the one-to-one mapping will be involved in profile propagation. Consistent procedures are procedures with the same control flow structure in both versions. Code block matching is straightforward for these procedures. Restructured procedures are procedures with different control flow structure in the two versions. Average sizes of procedures are calculated for the new version.

<table>
<thead>
<tr>
<th>Program Versions</th>
<th>cc1 2.8.0</th>
<th>cc1 2.8.1</th>
<th>cc1 2.95</th>
<th>ghostscript 5.0</th>
<th>ghostscript 5.10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Procedures</td>
<td>2314</td>
<td>2318</td>
<td>2874</td>
<td>3759</td>
<td>4269</td>
</tr>
<tr>
<td>Procedures Included in One-to-one Mapping</td>
<td>2314</td>
<td>2057</td>
<td></td>
<td></td>
<td>3746</td>
</tr>
<tr>
<td>Consistent Procedures</td>
<td>1800</td>
<td>1122</td>
<td></td>
<td></td>
<td>2835</td>
</tr>
<tr>
<td>Average Size of Consistent Procedures</td>
<td>11.0 Code Blocks</td>
<td>7.2 Code Blocks</td>
<td></td>
<td></td>
<td>5.6 Code Blocks</td>
</tr>
<tr>
<td>Restructured Procedures</td>
<td>514</td>
<td>935</td>
<td></td>
<td></td>
<td>911</td>
</tr>
<tr>
<td>Average Size of Restructured Procedures</td>
<td>71.1 Code Blocks</td>
<td>48.7 Code Blocks</td>
<td></td>
<td></td>
<td>21.8 Code Blocks</td>
</tr>
</tbody>
</table>

### 5.4.3. Code Block Matching Results

For each code block $x$, we may successfully find one matching block that according to our static analysis is most likely equivalent to $x$ in terms of execution pattern. We call this a confident match. For other blocks, which are involved in relatively significant code changes, we have to find fuzzy matches and possibly conversion formulas. For each procedure, we define its Matching Confidence Rate as the percentage of code blocks for which we find confident matches. While the Matching Confidence Rate of a procedure does reflect the effectiveness of our matching algorithm, a more important factor is how much the procedure has changed. Procedures that undergo significant changes are likely to have low Matching Confidence Rate.

Figure 5.8 (see next page) shows, for each test case, the cumulative distribution of Matching Confidence Rate among all restructured procedures. Note that consistent procedures, for which the Matching Confidence Rate is always 100%, are excluded from the graph. The results show that our algorithm produces high Matching Confidence Rates for most restructured procedures. In cc1 2.8.1 vs. 2.95, the worst among the three cases, over half of restructured procedures have Matching Confidence Rate of 85% or above.

Another observation is that the overall confidence of matching is lower for the cc1 2.8.1 vs. 2.95 test case than for the cc1 2.8.0 vs. 2.8.1 case. For the former case, only 39% of restructured procedures (363) have Matching Confidence Rate of 90% or higher. Together with 1122 consistent procedures, there are 1485 procedures registering 90% or above confidence level, a little over half among a total of 2874 procedures in cc1 2.95. For cc1 2.8.0 vs. 2.8.1, 75% of restructured procedures (384) have Matching Confidence Rate over 90%. Together with 1800 consistent procedures, there are 2184 procedures matched with high confidence out of 2318 procedures in cc1 2.8.1, a ratio as high as 94%. This is yet another evidence that the program
change between cc1 2.8.1 and 2.95 is much more significant than that between cc1 2.8.0 and 2.8.1. Calculated over all restructured procedures in the program, the cc1 2.8.1 vs. 2.95 test case has an average Matching Confidence Rate of 81%, compared to 92% for cc1 2.8.0 vs. 2.8.1. For the ghostscript test case, the average over all restructured procedures is 87%.

5.4.4. Profile Propagation Results

This section evaluates results from our experiments with profile propagation. Here we are not attempting to solve the general problem of determining whether one profile is “better” than another and by how much. Instead, our goal is that the propagated profile predicts the behavior of the new program version in the same way that a directly collected profile would. Using binary matching results, we propagate a profile collected on the old version to the new version, then compare the propagated profile with a profile collected directly on the new version using the same training workload that produced the original profile. The more similar our propagated profile is to the directly collected profile, the more accurate we say it is.

As a representative example, we use basic block execution count profiles in our experiments. In addition to comparing the two profiles over the whole binary program, we also perform partial profile comparison for a selected group of procedures that directly reflect the effectiveness of our profile propagation.

Figure 5.8. Cumulative distribution of Matching Confidence Rate. The y-axis shows the percent of restructured procedures with a Matching Confidence Rate higher than or equal to the x-axis value. A Matching Confidence Rate of 100% means the algorithm found a confident match for every code block in that procedure despite any changes in control flow structure. Note that this does not guarantee all the block matches are correct.
The procedures not covered by the one-to-one mapping are not involved in profile propagation, and thus not included in the partial profile comparison. We also exclude all consistent procedures from the partial profile comparison, as our binary matching will always produce a code mapping that preserves the exact information for these procedures in the old profile. This is the correct choice within the scope of intra-procedural analysis. Our tests show that a small percentage of consistent procedures do register different profile information in profiles collected directly on the new version, most likely due to inter-procedural code interactions. These differences are mostly minor, and cannot be predicted using intra-procedural analysis alone.

Some restructured procedures may not be touched at all during the execution of the training workload, and therefore have zero execution counts for all code blocks. Since our profile propagation never converts a zero count to non-zero, it will maintain all-zero execution counts for these procedures. Again, this is correct based on intra-procedural analysis, though changes in other procedures could cause some of these procedures to have non-zero execution counts in a directly collected profile for the new version. Table 5.4 summarizes the number of consistent procedures, restructured structures with all-zero execution counts, and restructured structures with at least one non-zero counts. Since profile propagation for the first two categories is

<table>
<thead>
<tr>
<th>Program Versions</th>
<th>cc1 2.8.0</th>
<th>cc1 2.8.1</th>
<th>cc1 2.95</th>
<th>ghostscript 5.0</th>
<th>ghostscript 5.10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Procedures</td>
<td>2314</td>
<td>2318</td>
<td>2318</td>
<td>2874</td>
<td>3759</td>
</tr>
<tr>
<td>Training Workload</td>
<td>24 input files for gcc from SPEC95</td>
<td>8 conversion tasks</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consistent Procedures</td>
<td>Total 1800</td>
<td>1122</td>
<td>2835</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equal Profile</td>
<td>1782</td>
<td>956</td>
<td>2684</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Different Profile</td>
<td>18</td>
<td>166</td>
<td>151</td>
<td></td>
<td></td>
</tr>
<tr>
<td>All-Zero Restructured Procedures</td>
<td>Total 299</td>
<td>517</td>
<td>632</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Equal Profile</td>
<td>299</td>
<td>500</td>
<td>623</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Different Profile</td>
<td>0</td>
<td>17</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Uncertain Procedures</td>
<td>215</td>
<td>418</td>
<td>279</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 5.4. Summary of profiles.** We use the ATOM instrumentation tool to collect basic block execution count profiles. The profile for each version is combined from all training scenarios by a simple addition of execution counts. This table divides the procedures into three categories: consistent procedures, restructured procedures with all-zero execution counts, and restructured procedures with at least one non-zero counts (i.e. uncertain procedures). We further divide the first two categories into procedures that have exactly equal profiles in the old and new versions and those that have different profiles. For restructured procedures, two all-zero profiles are considered equal even if they have different numbers of code blocks, while an all-zero profile and a non-zero profile are considered different.
straightforward and does not reflect the design of our binary matching algorithm, our profile comparison focuses on restructured procedures with non-zero execution counts, or what we call *uncertain procedures*. As shown in Table 5.4, uncertain procedures make up a relatively small portion of the program (between 7% and 15% in our test cases).

We use several metrics from Chapter 3 for our comparison of the propagated profile and the directly collected profile. For a detailed analysis, we perform comparison on a per-procedure basis as well as for the whole program. Note that before we compare the profiles, we normalize each procedure by dividing the execution counts for all blocks by the execution count of the first block. This allows us to use scale-sensitive metrics on procedures that have the same profile information on a different scale due to inter-procedural code interactions. Here is a review of the metrics we use in this section:

*Equal Value Percentage (E.V. Percentage)*: The percentage of code blocks in a procedure (or the whole program) that have equivalent profile values in both profiles.

*Similar Value Percentage (S.V. Percentage)*: The percentage of code blocks in a procedure (or the whole program) that have similar profile values in both profiles. Somewhat arbitrarily we set the threshold to 2.0, which means we consider two values to be similar if neither one is more than twice as large as the other one.

*Correct Coverage Percentage (C.C. Percentage)*: The percentage of code blocks in a procedure (or the whole program) that either have non-zero execution counts in both profiles, or zero counts in both. In other words, the two profiles agree on whether these code blocks were ever “covered” by the program execution.

*Max-to-Min Average (M.-M. Average)*: The geometrical mean of the relative ratio between two profile values for each code block in a procedure (or the whole program). Note that when calculated over the whole program, the ratio is averaged among all code blocks in the program in one step, not averaged within each procedure and then among the procedures.

*Key Matching Function*: For any integer $n$, it counts the number of code blocks that belong to the top $n$ entries of both profiles, i.e., the top $n$ most frequently executed code blocks. This metric is applied only to individual procedures and not to the whole program, as most optimizations do not compare code block execution counts across procedure boundaries.

The first three metrics, *E.V. Percentage, S.V. Percentage* and *C.C. Percentage*, exemplify three different levels of accuracy we expect from the propagated profile. *E.V. Percentage* is extremely strict as it reaches 100% only when the two profiles are exactly the same. Since a profile-based optimizer only uses certain information extracted from the profile, it would not require the propagated profile to be exactly the same as an accurate profile in order to achieve the same performance benefit. *S.V. Percentage* and *C.C. Percentage* are less strict, but still penalize any discrepancy between a zero and a non-zero profile value. Given specific optimizations, we can design different metrics that reflect the type of information we need from the profile.

For all three test cases, Figure 5.9 (see next page) shows the cumulative distribution of *E.V. Percentage, S.V. Percentage* and *C.C. Percentage* as applied to each procedure. For $cc1$ 2.8.0 vs. 2.8.1, the results are very promising. The propagated profile matches the directly collected profile perfectly for 36% of uncertain procedures (77), and achieves correct rate above 90% for 72% of them (153). Together with consistent procedures and all-zero restructured
Figure 5.9. Partial profile comparison: Cumulative distribution of three metrics. *Uncertain procedures* are restructured procedures with non-zero execution counts. The X-axis gives a threshold value for the metrics, while the Y-axis shows the percentage of uncertain procedures that score the given value or higher on each metric.
procedures, the propagated profile provides 100% accurate profile information for 2158 procedures and over 90% accurate profile for 2236 procedures, representing 93% and 97% of all procedures in cc1 2.8.1, respectively. Among the remaining procedures, all but one score over 60% in Similar Value Percentage. If we only need information about which code blocks are executed, our profile propagation produces 100% accurate information for 57% of uncertain procedures and over 90% accurate for 88% of them.

The results for cc1 2.8.1 vs. 2.95 are less impressive. However, profile propagation still produces 100% accurate information for 16% of uncertain procedures (67). Over the entire program cc1 2.95, we are provided with 100% accurate profile information for 1522 procedures, or 53% of all procedures. The results for ghostscript 5.0 vs. 5.10 fall in between those for the two cc1 cases.

The three metrics shown in Figure 5.9 indicate the percentage of code blocks in a procedure that are correctly matched, but not how accurate the rest of the matches are. Max-to-Min Average gives us a numerical ratio that takes all code blocks into account. Figure 5.10 shows the per-procedure results for this metric. Once again, cc1 2.8.0 vs. 2.8.1 yields very good results. 80% of uncertain procedures have M.-M. Average lower than 1.5, while the largest M.-M. Average among all procedures is less than 10. For both cc1 2.8.1 vs. 2.95 and ghostscript 5.0 vs. 5.10, about 80% of uncertain procedures have M.-M. Average lower than 4.0, but several procedures have M.-M. Average larger than 100. Manual inspection reveals that these cases are caused by a number of wrong matches for code blocks with very large execution counts.

![Figure 5.10. Partial profile comparison: Cumulative distribution of M.-M. Average.](image)

Each line shows the percentage of uncertain procedures with Max-to-Min Average lower than or equal to the X-axis value.
<table>
<thead>
<tr>
<th>Program Versions</th>
<th>cc1</th>
<th>cc1</th>
<th>ghostscript</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.8.0</td>
<td>2.8.1</td>
<td>2.95</td>
</tr>
<tr>
<td>E.V. Percentage</td>
<td>88.4%</td>
<td>61.6%</td>
<td>71.1%</td>
</tr>
<tr>
<td>S.V. Percentage</td>
<td>95.3%</td>
<td>82.3%</td>
<td>90.3%</td>
</tr>
<tr>
<td>C.C. Percentage</td>
<td>96.8%</td>
<td>87.2%</td>
<td>93.8%</td>
</tr>
<tr>
<td>M.-M. Average</td>
<td>1.23</td>
<td>2.30</td>
<td>1.65</td>
</tr>
<tr>
<td>E.V. Percentage</td>
<td>95.9%</td>
<td>82.2%</td>
<td>92.7%</td>
</tr>
<tr>
<td>S.V. Percentage</td>
<td>98.4%</td>
<td>90.9%</td>
<td>97.3%</td>
</tr>
<tr>
<td>C.C. Percentage</td>
<td>98.9%</td>
<td>92.6%</td>
<td>97.9%</td>
</tr>
<tr>
<td>M.-M. Average</td>
<td>1.07</td>
<td>1.61</td>
<td>1.17</td>
</tr>
</tbody>
</table>

**Table 5.5. Overall results for partial and complete profile comparison.** This table lists the four metrics as they are calculated over all uncertain procedures (i.e. partial comparison) and over all procedures in the new version (i.e. complete comparison), not on a per-procedure basis but across procedure boundaries.

Table 5.5 lists the results of the above four metrics as calculated over all uncertain procedures and over the whole binary program. In order to do this calculation, we need to generate profile information for procedures that are not included in our binary matching process. Most of these procedures are added between the old version and the new version. In our experiments, we simply create all-zero execution counts for these procedures. Our limited experience suggests that these new procedures do tend to have all-zero execution counts when the new version is used.

Finally, we use the key matching method to evaluate how accurately the propagated profile pinpoints the most frequently executed part of the code. This measure is important to optimizations such as code placement, instruction scheduling and path-based optimization. We conduct our test among uncertain procedures that contain more than 10 code blocks. Table 5.6 lists the percentage of tested procedures that satisfy the given criteria on key matching results.

<table>
<thead>
<tr>
<th>Program Versions</th>
<th>cc1</th>
<th>cc1</th>
<th>ghostscript</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.8.0</td>
<td>2.8.1</td>
<td>2.95</td>
</tr>
<tr>
<td>Number of Tested Procedures</td>
<td>196</td>
<td>314</td>
<td>152</td>
</tr>
<tr>
<td>Percentage of Tested Procedures that Satisfies</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( K.M.(1) = 1 )</td>
<td>92.9%</td>
<td>81.8%</td>
<td>84.2%</td>
</tr>
<tr>
<td>( K.M.(5) = 5 )</td>
<td>73.5%</td>
<td>37.9%</td>
<td>48.7%</td>
</tr>
<tr>
<td>( K.M.(10) = 10 )</td>
<td>58.2%</td>
<td>26.1%</td>
<td>27.6%</td>
</tr>
<tr>
<td>( K.M.(10) &gt; 5 )</td>
<td>92.3%</td>
<td>65.6%</td>
<td>79.6%</td>
</tr>
</tbody>
</table>

**Table 5.6. Key matching test results.** We only tested uncertain procedures with more than 10 code blocks. For each procedure, \( K.M.(n) \) measures how many of the top \( n \) most frequently executed code blocks in the propagated profile is also among the top \( n \) in a directly collected profile.
In the case of cc1 2.8.0 vs. 2.8.1, the propagated profile correctly identifies the most frequently executed code block in over 90% of tested procedures. For a similar percentage of procedures it makes at least six correct picks out of the top 10 code blocks. The propagated profile also correctly identifies the entire set of top 10 for almost 60% of tested procedures. The results are not as good for the case of cc1 2.8.1 vs. 2.95, with the propagated profile selecting the top 10 correctly for only about a quarter of tested procedures. However, it identifies more than half of the top 10 for almost two-thirds of tested procedures. For over 80% of them, the propagated profile agrees with the directly collected profile about the single most frequently executed code block. The results for ghostscript are similar to, but somewhat better than, the results for cc1 2.8.1 vs. 2.95.

Summary

This chapter presented a control flow-based binary matching algorithm, supported by some promising results from our experiments. Our results show that when the two versions are not drastically different, we can generate propagated profile that closely resembles a directly collected profile by many measures. If there are significant changes between two versions, profile propagation may not be able to generate high-quality profile for the entire new version. However, it can still provide accurate information for a considerable portion of the program.
Chapter 6. Progressive Profiling

When we investigated profile propagation in the previous two chapters, we focused on the accuracy of propagated profiles. In the context of progressive profiling, however, accuracy is not the only measure of success, as one of our major goals is to reduce the time needed for profiling runs in order to generate high-quality profiles. This chapter discusses the process of progressive profiling over multiple generations of a program and how it can reduce profiling time substantially while maintaining high accuracy of profile. The time saving comes from conducting selective profile collection instead of re-generating the entire profile. The difference in profiling time can more than compensate for the time spent on profile propagation.

We built our prototype system of progressive profiling on top of our control flow-based binary matching tool on the DEC Alpha platform. Using this prototype system, we explore different options in conducting progressive profiling. While either of our binary matching projects can serve as a base for a progressive profiling system, we could not carry out similar experiments on the x86 architecture because we no longer had access to the system where we built BMAT.

6.1. Methodology Choices in Progressive Profiling

Profile propagation introduces the possibility of combining propagated profile information with directly collected profile information. As discussed in previous chapters, our profile propagation cannot produce profile information for procedures that we fail to recognize. In addition, it may produce inaccurate information for procedures that have changed drastically or are affected by changes in other procedures. Selective profile collection can address these problems by generating new profile information for some procedures as we deem necessary. The newly collected information is combined with the propagated profile to produce a hybrid profile for the new version.

This process introduces a trade-off between profile propagation and selective profile collection. If we use more propagated information and less directly collected information, we achieve higher reduction in profiling time, but the result profile may become less accurate due to significant program changes and inter-procedural code interactions that are not modeled in our binary matching. On the other hand, if we are conservative in using propagated information, we have to spend more time on collecting new profile information.

In our system model, we consider three major dimensions of choice in progressive profiling. The rest of this section discusses them in detail.

6.1.1. Choices in Training Workload Execution

In current practice, since the profiling process is repeated for each new program version, a fixed training workload is often used for all versions. By preserving profile information across program revisions, progressive profiling offers the option of running different training workloads for each version and using profile propagation to generate an accumulated profile. This accumulated profile reflects the program behavior under all the training workloads combined. Therefore, we can reduce total profiling time by dividing the set of training workload we need to run over multiple versions.
6.1.2. Choices in Profile Propagation

For profile propagation, we propose three models that differ in the amount of information they choose to propagate.

Maximum Propagation

In binary matching, we try to find matches for all code blocks in all procedures included in our one-to-one mapping. Using the binary matching results, we can propagate profile information for all those procedures, regardless of whether the block matches are correct. We call this model maximum propagation.

Confident Propagation

If we exclude procedures for which the propagated information will not be accurate, we increase the overall accuracy of the propagated profile. In reality, we cannot calculate the accuracy of propagated profile without first generating a directly collected profile, which is exactly what we are trying to avoid with progressive profiling. Therefore, we use our binary matching confidence for each procedure as a predictor for the accuracy of propagated profile. As defined in Section 5.4.3, the Matching Confidence Rate for a procedure is the percentage of code blocks for which we find a confident match. Figure 6.1 suggests that there is a correlation between the Matching Confidence Rate of a procedure and the accuracy of propagated profile for that procedure. In the confident propagation model, we conduct profile propagation for a procedure only if its Matching Confidence Rate surpasses a pre-selected threshold. Confident propagation reduces the amount of inaccurate profile information in the propagated profile, but also increases the amount of information we need to collect after the program revision.

![cc1 2.8.1 vs. 2.95 (418 uncertain procedures)](image)

Figure 6.1. Correlation between procedure matching confidence and accuracy of propagated profile. This scattered plot shows a correlation between higher matching confidence and higher accuracy of propagated profile. For each procedure, Matching Confidence Rate is the percentage of code blocks with confident matches (see Section 5.4.3). The accuracy of propagated profile for each procedure is measured with its Similar Value Percentage (threshold 2.0) against a directly collected profile.
An even more fine-grained method would be to run profile collection only for portions of procedures that are not matched with high confidence. That method is not investigated in this thesis.

**Basic Propagation**

This is a base model that does not employ detailed binary matching. As discussed in Section 5.4.2, procedures that have identical control flow structure in both program versions, which we call consistent procedures, keep their profile information as it is in our profile propagation. If we limit profile propagation to consistent procedures, we can save time by eliminating intra-procedural code block matching. However, the re-use of profile information would be limited to consistent procedures which, as shown in Table 5.3, tend to be the smaller procedures in the program. This model, *basic propagation*, is the same profile propagation method used in the Spike system [Cohn et al. 1997].

Similar to Spike, we identify consistent procedures by calculating a control flow signature for each procedure in each version. If a procedure has the same signature in both versions, we copy its profile information in the old version onto the new version.

**6.1.3. Choices in Profile Collection**

Once we have a propagated profile for the new version, we can conduct additional profile collection to augment the profile as needed. If short on time, we may even skip the profile collection altogether, but as we have discussed the propagated profile may not describe all procedures in the new version. The method of additional profile collection is related to whether we use the same training workload for each version.

**Accumulative Collection**

If we use different training workloads for each version, the propagated profile is all the information we have regarding the workload run on the old version. To maximize the total profile information, we conduct complete profiling on the new version with the new training workload, and combine the directly collected profile with the propagated profile. We call this *accumulative collection*. Ideally, the combined profile should be identical to a profile we would get by running both training workloads directly on the new version. In our model, however, a newly added procedure does not carry any profile information that reflects the old training workload. This causes imbalance between the richness and diversity of profile information for different procedures. The consequences of this imbalance are yet to be investigated.

The accumulative model also highlights two methodology issues in progressive profiling. One is how to combine profiles from different sources such as different workloads. Some researchers have investigated different methods of combining profiles [Fisher and Freudenberger 1992, Savari and Young 1999]. In our experiments, we use a simple addition of execution counts to combine a propagated profile and a directly collected profile, or in fact any two profiles for the same program version. For accumulative collection, this method makes it easy to compare the result profile with a profile collected directly on the new version using all the training workloads involved in the process. The second methodology issue is how to age old profile information over time. This issue has not been carefully studied, and is beyond the scope of this thesis.
Fill-in Collection

If we use the same training workload for each version, the propagated profile makes it unnecessary to run profile collection for the entire new version. Instead, we collect new profile information only for procedures that are not included in the profile propagation. This new information is merged with the propagated profile to produce a new profile that covers all procedures in the new version. This *fill-in collection* model minimizes required profile collection time under the condition that the final profile covers all procedures.

Smart Fill-in Collection

*Smart fill-in collection* is an adaptive approach that is based on *fill-in collection* but with two modifications that target extreme cases. First of all, if only a small number of procedures need to be profiled, we skip selective profile collection in order to save the time required for instrumenting the program and executing the training workload. In our implementation, we skip the profile collection step when the procedures that need profile information make up for 0.1% or less of all code blocks in the program.

The second modification is for cases where two program versions are significantly different. Results in Section 5.4.4 show that in such cases, such as `cc1 2.8.1 vs. 2.95`, the accuracy of the propagated profile becomes relatively low. Smart fill-in collection addresses this by abandoning profile propagation and going back to re-generating the entire profile for the new version. In doing so, we do not save any profiling time, and in fact spend extra time on binary matching. However, over multiple generations of the program we can still save total profiling time while maintaining high accuracy of profile. In practice, this simply means that we preserve profile information across minor revisions but restart the profile collection from scratch after any major revision. To assess how different two program versions are, we use statistics from the binary matching process. Possible measurements include the percentage of procedures that are not included in binary matching and the distribution of matching confidence rate among procedures. In this implementation, we always finish the entire binary matching process. A more adaptive approach would be to stop the binary matching once we conclude that the two versions are significantly different.

If neither of the two extreme conditions hold, we perform profile propagation and selective profile collection according to the *fill-in collection* model.

6.2. A Case Study

In this section we use a realistic example to demonstrate different models of progressive profiling. With each model we simulate a profiling process spanning five versions of `cc1`, from 2.8.0 to 2.95.2. The profiling process involves profile collection on each version and profile propagation between each two consecutive versions. Our training workload consists of 24 input files for the `gcc` benchmark in SPEC95 suite. We use six selected models to cover the three sets of choices discussed in the last section.

1. **Traditional**: Traditional profiling model with no profile re-use. There is no profile propagation, and complete profile collection is repeated for each version.

2. **Basic**: Basic propagation and fill-in collection.
3. **Confident**: Confident propagation (with a threshold of 80% for Matching Confidence Rate) and fill-in collection.

4. **Maximum**: Maximum propagation and fill-in collection.

5. **Smart**: Maximum propagation and smart fill-in collection.

6. **Divided**: For the previous five models, the entire set of training workload is used for all versions. For the Divided model, the training workload is randomly divided into five groups of 5, 5, 5, 5 and 4 input files and executed on five versions separately. This model uses maximum propagation and accumulative collection.

   Note that we always perform complete profile collection for the first version. For each model, the result profile for the last version, cc1 2.95.2, is compared to a profile collected directly on that version using the entire set of training workload. Table 6.1 (see next page) presents the profiling time measurements and profile comparison results, while Figure 6.2 that follows illustrates the same data in a graph format.

   With only basic propagation, we reduce the total profiling time by 34% and obtain a highly accurate profile. The Confident and Maximum models demonstrate the trade-off between profiling time and profile quality. By propagating profile information for more procedures, we save more profiling time while sacrificing some profile quality. However, even with maximum propagation where we save 58% in profiling time, we achieve Similar Value Percentage of 92.4% and K.M.(10) Average of 8.99. In other words, 92.4% of all code blocks have execution counts that fall between 50%-200% of their accurate execution counts, and on average we correctly identify nine out of ten most frequently executed blocks in each procedure (excluding procedures with ten blocks or less).

   Smart fill-in collection proves to be very successful when coupled with maximum propagation. By restarting the profile collection with cc1 2.95, we eliminate the inaccuracy caused by the significant changes between cc1 2.8.1 and 2.95. Since there are only minor changes from cc1 2.95 to cc1 2.95.2, our result profile turns out to be highly accurate. Meanwhile, we eliminate the profile collection steps for cc1 2.8.1 and cc1 2.95.2, which compensates for the long profile collection for cc1 2.95. As a result, we cut the total profiling time by more than half and sacrifice little profile quality.

   The Divided model saves the most amount of profiling time, as the entire training workload only needs to be profiled once. The quality of the result profile is not as good as maximum propagation with smart fill-in collection, but better than maximum propagation with fill-in collection. This shows that by executing the training workload in parts over multiple generations of the program, we achieve huge reduction in profiling time while still generating profile of good quality.

   We can also tell from Table 6.1 that the percentage saving in total profiling time is affected by the relation between the time needed for profile collection and for profile propagation. In our example, the profile collection time for the entire training workload is less than ten times the time for profile propagation between two versions. In realistic situations, the training workload may be much larger, enabling even bigger savings in total profiling time.
<table>
<thead>
<tr>
<th>Model</th>
<th>Name</th>
<th>Traditional</th>
<th>Basic</th>
<th>Confident</th>
<th>Maximum</th>
<th>Smart</th>
<th>Divided</th>
</tr>
</thead>
<tbody>
<tr>
<td>training workload</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>all</td>
<td>divided</td>
</tr>
<tr>
<td>profile propagation</td>
<td>none</td>
<td>basic</td>
<td>confident</td>
<td>maximum</td>
<td>maximum</td>
<td>maximum</td>
<td></td>
</tr>
<tr>
<td>profile collection</td>
<td>complete</td>
<td>fill-in</td>
<td>fill-in</td>
<td>fill-in</td>
<td>smart fill-in</td>
<td>accumul.</td>
<td></td>
</tr>
</tbody>
</table>

| (c) cc1 2.8.0        | 116.4        | 116.4       | 116.4  | 116.4     | 116.4   | 22.3  |
| (p) 2.8.0⇒2.8.1      | --           | 11.0        | 21.6   | 21.6       | 21.6    | 21.6  |
| (c) cc1 2.8.1        | 117.3        | 93.4        | 27.5   | 23.5       | 0.0     | 41.8  |
| (p) 2.8.1⇒2.95       | --           | 7.5         | 18.0   | 18.0       | 16.2    | 18.0  |
| (c) cc1 2.95         | 152.8        | 137.2       | 85.5   | 49.5       | 152.8   | 32.7  |
| (p) 2.95⇒2.95.1      | --           | 7.7         | 15.7   | 15.7       | 15.7    | 15.7  |
| (c) cc1 2.95.1       | 152.6        | 30.3        | 0.0    | 0.0        | 0.0     | 30.2  |
| (p) 2.95.1⇒2.95.2    | --           | 7.8         | 16.2   | 16.2       | 16.2    | 16.2  |
| (c) cc1 2.95.2       | 154.0        | 46.0        | 31.7   | 31.1       | 0.0     | 27.0  |
| (c) Subtotal         | 693.1        | 423.3       | 261.1  | 220.5      | 269.2   | 154.0 |
| (p) Subtotal         | --           | 34.0        | 71.5   | 71.5       | 69.7    | 71.5  |
| Total                | 693.1        | 457.3       | 332.6  | 292.0      | 338.9   | 225.5 |
| % Saving             | --           | 34%         | 52%    | 58%        | 51%     | 67%   |
| E.V. Percentage      | 100.0%       | 98.4%       | 90.0%  | 83.9%      | 99.6%   | 83.7% |
| S.V. Percentage      | 100.0%       | 99.7%       | 96.1%  | 92.4%      | 99.9%   | 95.3% |
| C.C. Percentage      | 100.0%       | 99.7%       | 97.2%  | 94.4%      | 100.0%  | 97.1% |
| M.-M. Average        | 1.00         | 1.02        | 1.20   | 1.44       | 1.00    | 1.20  |

(c) : profile collection; (p) : profile propagation

Table 6.1. Profiling time and quality of final profile for six different models of progressive profiling. All execution times are measured with the `time` utility in the DEC OSF operating system. Each value is the average of five separate measurements. Standard deviation is under 0.2 second for all time values under 100 seconds and under 0.5 second for all others. % Saving in profiling time is compared against the Traditional Model. Quality of propagated profile is calculated over all code blocks in the binary program, except K.M.(10) Average is the average value of K.M.(10) among all procedures with more than 10 code blocks.
Figure 6.2. Profiling time and quality of final profile for six different models of progressive profiling.

Summary

This chapter discusses and demonstrates some different models of progressive profiling. Our results show that progressive profiling can reduce total profiling time by over a half while still producing profiles of high quality. The results also indicate that various choices in progressive profiling present a trade-off between the amount of profiling time saved and the quality of the final profile. The Smart model proves that adaptive methods, even simple ones, can help us achieve better results. More sophisticated adaptive methods are an important subject for future research. The Divided model highlights an opportunity provided by progressive profiling, i.e., the spreading of training workload over multiple generations of a program.
Chapter 7. Conclusions

Profile-Based Optimization (PBO) is a relatively new class of program optimization that can lead to significant performance improvement, especially for modern applications that are large and interactive. Many specific types of PBO have been implemented, but there has been little research on the issues that connect the theory of PBO to its effective use in commercial practice. Specifically, it is not clear how we can generate high-quality profiles that are necessary for realizing PBO’s potential performance benefits.

A critical problem with applying PBO in software production environments is the lack of time for profile collection due to frequent program revisions. Progressive profiling is a new profiling methodology that addresses this problem by preserving profile information even when a program undergoes changes. By doing so, progressive profiling prolongs the shelf life of profile information and allows the accumulation of information over multiple generations of a program.

Progressive profiling is based on the methods of profile propagation and selective profile collection. In this thesis we use binary matching to drive both processes, using no additional information from program source code, programmers or compilers. Given an old version and a new version of a program, we use static analysis to produce a mapping between the two binaries. Based on this mapping we perform profile propagation, which converts an existing profile for the old version to a new profile that describes the new version. If changes between the two versions are small, the propagated profile can provide highly accurate prediction for the behavior of the new version, and therefore serves as a good replacement for profiles collected directly on the new version. If due to significant program changes our profile propagation cannot produce high-quality profile information for parts of the program, we can conduct selective profile collection just for those parts. This still saves time over the traditional model where new profiles have to be re-generated from scratch.

We presented two different binary matching algorithms implemented on two different platforms. Both algorithms first identify a one-to-one mapping between the procedures in two program versions, then perform code matching within each pair of corresponding procedures. One algorithm is mostly based on the program’s code contents, while the other one is mostly based on its control flow structure. These two approaches, though both designed for general-purpose binary matching, may be suitable for different platforms, different benchmark programs and different coding conventions. A direct comparison between the two would require first implementing them on the same platform. We did not have the time and setup to do so for this thesis, but hope to carry out such a comparison in the future.

For the purpose of simplicity and flexibility, both binary matching algorithms contain a number of heuristics and adjustable parameters. The tuning of these heuristics and parameters may depend on the instruction set architecture, programming language conventions, compiler conventions, and programming style. Our experimental results showed that with a moderate amount of effort, the algorithms work well on the popular application programs we used as benchmarks. How to tune the algorithms so that they work well under different situations is an important area for future research.

To evaluate the effectiveness of our profile propagation, we compared propagated profiles with profiles collected directly on the new program version. We introduced a set of old and new metrics for the quantitative comparison of profiles. We did not focus on comparing the
optimization results using different profiles, because such a comparison is greatly influenced by the specifics of the optimization we use. In our experiments with real-world application programs, the propagated profiles closely resemble the directly collected profiles.

For our code contents-based algorithm on the Intel x86 architecture, we showed that across a set of large DLLs from Microsoft Windows 2000 with two versions that are six weeks apart, our propagated profiles were almost as effective as directly collected profiles on two measures. When used to predict branch biases, the success rates of propagated profiles were on average 98.8% as high as those of directly collected profiles. For identifying which program basic blocks will be executed under the testing workload, the correct rates of propagated profiles averaged a 97.7%. For our control flow-based algorithm on the DEC Alpha architecture, we used cc1, the front-end module of the GNU C compiler gcc, as a major benchmark program. For two versions that were released two months apart, our propagated profile provided 100% accurate basic block execution counts for 2158 out of 2314 procedures, and averaged a correct rate of 86% for the other 156 procedures. For two other versions that were separated by over a year, the accuracy of propagated profile was much lower.

We built a prototype system that performs the complete process of progressive profiling. Using this system we examined six different models of progressive profiling and demonstrated their effectiveness over five versions of cc1. Our experiments showed that progressive profiling can reduce the total profiling time by more than half while still producing a highly accurate profile. The percentage reduction in profiling time would be even higher if the training workload is extremely large, which is often the case in commercial situations. This suggests that it is plausible to obtain high-quality profiles in a software production environment without spending an unrealistic amount of time on program profiling runs.

Our results also suggested that the trade-off between profile propagation and selective profile collection has a significant impact on the overall performance of progressive profiling. One model that takes an adaptive approach provided notable improvement over similar but non-adaptive models, suggesting that there is great potential for more sophisticated progressive profiling models.

Progressive profiling is only a first step into a largely unexplored area that is the methodology of applying PBO. In order to make PBO a practical tool that can be used broadly and effectively in the software industry, we still need to resolve many important issues concerning how to collect, evaluate and manage profiles so that they can lead to effective optimizations. These issues have been largely ignored by most research systems, yet they are crucial in realistic situations. Some of these issues are now introduced or highlighted by progressive profiling. For example, how would we select profiling methods and design the profile collection process so that the profile we get is most representative of how the program will be used? How would we combine profiles from different sources, such as propagated profiles and directly collected profiles or profiles from different runs? How would we manage these profiles as time passes? Also, how would we evaluate collected profiles and decide when to trigger profile-based optimizations? We believe that many methods and metrics presented in this thesis can also be applied to future research on these topics.
References


