Journal of Instruction-Level Parallelism 2 (2000) 1-6 Submitted 2/00; published 4/00

BMAT -- A Binary Matching Tool for Stale Profile Propagation

Zheng Wang ZHWANG@EECS.HARVARD.EDU
Division of Engineering and Applied Sciences, Harvard University,
33 Oxford Street, Cambridge, MA 02138 USA

Ken Pierce KPIERCE@MICROSOFT.COM
Scott McFarling SMCFAR@MICROSOFT.COM
Microsoft Research,
One Microsoft Way, Redmond, WA 98052 USA

Abstract

A major challenge of applying profile-based optimization on large real-world applications is how to capture adequate profile information. A large program, especially a GUI-based application, may be used in a large variety of ways by different users on different machines. Extensive collection of profile data is necessary to fully characterize this type of program behavior. Unfortunately, in a realistic software production environment, many developers and testers need fast access to the latest build, leaving little time for collecting profiles. To address this dilemma, we would like to re-use stale profile information from a prior program build. In this paper we present BMAT, a fast and effective tool that matches two versions of a binary program without knowledge of source code changes. BMAT enables the propagation of profile information from an older, extensively profiled build to a newer build, thus greatly reducing or even eliminating the need for re-profiling. We use two metrics to evaluate the quality of the results using propagated profile information: static branch prediction and the accuracy of code coverage. These metrics measure how well the matching algorithm works for the frequently executed core code and across the whole program, respectively. Experiments on a set of large DLLs from Microsoft Windows 2000 and Internet Explorer show that compared to freshly collected profiles, propagated information using BMAT is typically over 99% as effective in branch prediction and over 98% as accurate in code coverage information.

1. Introduction

1.1. Stale Profile Propagation

In recent years, there has been a lot of research on various types of profile-based optimizations (PBO) and their potential benefits. Many of these studies make the assumption that an adequate set of profile data is available. In practice, it is often difficult to generate high quality profiles. In a real-world production environment, delivering a fresh build back to the development and testing teams can be a critical bottleneck. Therefore, the time available for collecting profiles is by necessity short. Meanwhile, the size and complexity of popular interactive applications continues to grow. Such applications can be used in a large variety of ways. Even for the same task, differences in the position of the mouse or the placement of a window can cause an entirely different code path to be executed. To adequately profile this vast space of potential usage in limited time is problematic. In addition, after an application is shipped, it may require a number of minor bug fixes and patches over a span of years. By this time, the original development team

with the expertise to conduct profile collection for the specific application may well have scattered to other projects.

These problems are results of the conventional way in which profiles are collected and used. Every time the program is altered, all previously collected profiles become *stale* and must be discarded. Therefore, new profiles must be created from scratch. In this paper, we propose a method of propagating and re-using stale profile information. Using a mapping between a previous build and the current build of a binary program, we convert the existing profiles for the previous build so that they describe expected behavior of the current build. Thus, profile collection for the previous build becomes part of the profiling process for the current build. In effect, this extends the time available for profile collection to days or even weeks, compared to the minutes available now. For minor changes, it may be possible to remove the profiling step completely.

1.2. Binary Matching

To propagate stale profile information, we need a mapping between the old and new builds of the program. For the propagated profile to accurately describe how the new build will behave, the mapping needs to match program sections that would be executed in the same manner in two versions. The more accurate the mapping is, the more effective the profile propagation will be.

BMAT is a binary matching tool we developed for this purpose. It compares two versions of a binary program and finds matches between their code and data blocks. Here a *match* refers to a link between a block in one version and a block in the other, indicating that the two blocks are the best equivalent of each other in terms of program execution. In BMAT, code blocks refer to program basic blocks, while data blocks are divided according to how data are accessed. We conduct code matching on the basic block level because most types of profile information that are gathered in current practice, such as execution counts and branch biases, are associated with basic blocks. The propagation of such profile information is straightforward once we have a mapping at the basic block level.

In realistic situations, the majority of program blocks can be expected to remain the same or undergo only minor changes between two program versions that are not too far apart in time. Therefore, we match the blocks mainly based on their contents. However, we also consider the program structure during the matching process. With certain types of program changes, blocks that appear to be similar or even identical may not be equivalent in terms of execution patterns, while two seemingly different blocks may be surrounded by identical control flow. For stale profile propagation, if the block matches produce a profile that accurately describes how the new version runs, we consider the matches to be *correct*.

To obtain a propagated profile that is complete, we need to find matches for as many blocks as possible. For equivalent blocks with small differences, we use a *fuzzy match* based on the hypothesis that they are used in the same or a similar way. For blocks that do not have obvious matches due to major program changes, we try to find matches based on both block contents and program structure. Note that the need to accommodate program changes may conflict with the goal of finding correct matches. One major reason is that due to certain characteristics of modern programming languages and popular programming styles, large applications tend to have many small code blocks that are similar or even identical. Therefore, an attempt to find a fuzzy match may also lead to incorrect matches.

1.3. Design Guidelines

In developing BMAT, we do not assume knowledge of source code changes. It is possible to use source code as hints in finding binary matches. However, in realistic situations, source code may not be available on the machine where profiling is conducted. Also, the association between the source code and binary level basic blocks may depend on how the source is compiled, adding complexity to the matching process. Some binary level basic blocks may not be visible at the source level.

Even though speed is not our top priority, we want BMAT to be 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.

Our implementation of BMAT is built on Windows NT for the x86 architecture. It uses the Vulcan binary analysis tool (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. Meanwhile, BMAT is designed for general-purpose binary matching regardless of the platform. The basic design of the matching algorithm is independent of the specifics of the x86 architecture. Therefore, porting BMAT should be straightforward. However, our choices of heuristics and the tuning of the heuristics are based on observations of common Windows NT applications for the x86 architecture. For a different platform, the heuristics may need to be adjusted based on the instruction set and the common application characteristics.

1.4. Contributions

Our work explores a fairly new and undeveloped area of profile-based optimizations. It makes the following contributions:

- We investigate the problem of obtaining adequate profile information for large and complex real world applications where only a limited time window is available for profile collection. We study the solution of propagating stale profile information from prior builds to the current build.
- We describe a sophisticated binary matching algorithm specifically designed for stale profile propagation. We choose the binary level for this process because of the practical difficulties of keeping source code and tracking source code changes on the build machine.
- We propose two metrics for evaluating the quality of propagated profile information: static branch prediction and code coverage. These metrics allow us to show that our algorithm is highly effective both on the frequently executed key blocks and across the full scope of the program.
- We conduct experiments on common Windows NT applications and show that stale profile propagation can extend the time window for profile collection to weeks or months.

1.5. Roadmap

The next section provides an overview of the algorithm we use in BMAT. Section 3 discusses the hashing-based matching method in detail. In Section 4, we present some results from applying BMAT to stale profile propagation for real-world applications. Section 5 examines some related work, and Section 6 concludes.

2. Algorithm Overview

In the rest of this paper, a *pair* of procedures (or blocks) refers to a set of two procedures (or blocks) taken from two versions of a binary program, one from each.

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. In many cases, this is simply because the rest of the program shifts in the address space. Such *indirect changes* may occur to control flow instruction targets, pointers, register allocation, etc.[1] In some extreme cases, indirect changes may cause two blocks that shouldn't match to appear identical while two blocks that should match appear different. Therefore, it is essential to detect and filter out indirect changes when looking for a match.

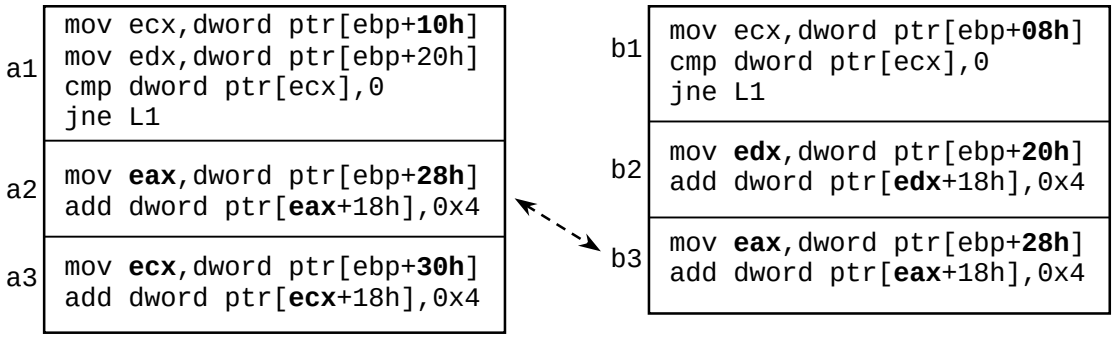

Figure 1. Example of Indirect Changes

In this example, a shift in the data layout of the program causes the address offsets in the `mov` instructions to change. A code change in the first block affects register allocation in the next two blocks. In addition, jump offsets in `a1` and `b1` may be different due to code movement. Note that due to indirect changes, blocks `a2` and `b3` are identical, while the correct matches are `a2-b2`, `a3-b3`. If we ignore the indirect changes in address offsets and register allocation, blocks `a2`, `a3`, `b2`, `b3` will all be identical, and correct matches can be made based on the relative positions of the blocks.

Our basic model is to divide the matching process into two stages. The first stage is to find a one-to-one mapping between the procedures in two versions based on their names, type information and code contents. Procedures that have been deleted or added will not be included. The mapping may also fail to include procedures that have changed drastically. During the second stage, we look for basic block matches within each pair of matched procedures. In other words, blocks in a procedure may only be matched to blocks in the corresponding procedure in the other version. Data blocks are matched in a separate phase from code blocks.

[1] In some literature, *direct* and *indirect* changes are called *primary* and *secondary* changes.

We confine code block matches within the procedure mapping to narrow down the problem space. To consider code block matches in the scope of the whole binary will increase both the complexity of the algorithm and the possibility of wrong matches. In reality, most code changes occur within procedure boundaries. There are cases where a procedure is split or multiple procedures are merged. Certain optimizations such as procedure inlining, if done differently for different builds, may also cause code movement across procedure boundaries. The current version of BMAT does not deal with these cases.

To compare basic blocks within a pair of procedures, we use a hashing-based algorithm. A 64-bit hash value is calculated for each block based on the opcodes and operands of its instructions. If a pair of blocks have the same hash value, it is likely that they constitute a match. If two or more blocks in one procedure have the same hash value, we use locality information and heuristic methods to identify which one makes the correct match.

The hashing calculation in our implementation is order-sensitive. In other words, two 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 paper.

As mentioned earlier, we need to deal with indirect changes when looking for matches. However, if we ignore all program elements that may be affected by indirect changes, we will have too little information left. Furthermore, BMAT is designed to find approximate matches for program blocks affected by direct changes. All these factors introduce a tradeoff between being flexible to look past minor changes and being precise to identify correct matches. In our design, the hashing-based matching algorithm includes multiple passes with different levels of matching fuzziness. This allows us to find more matches more accurately. A number of heuristic methods are used to help find the best match for each block.

Figure 2 is an overview of the whole matching process. The next three subsections present details of each phase.

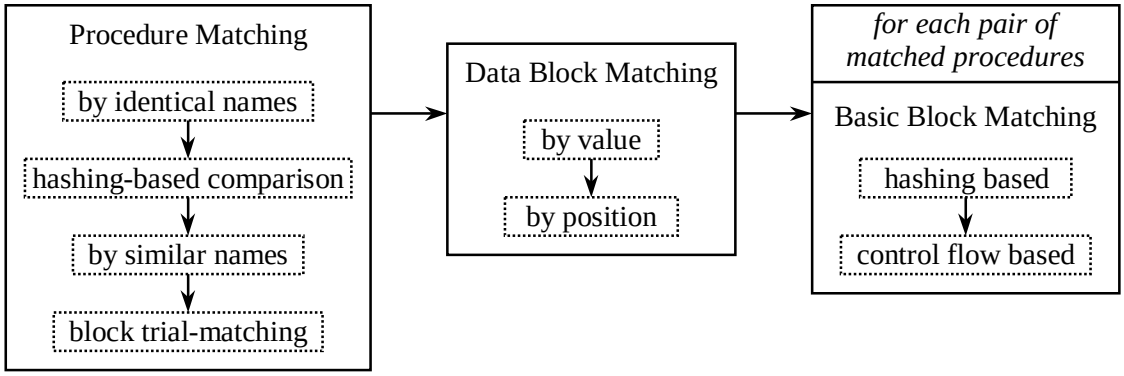

Figure 2. Overview of the Matching Process

2.1. Procedure Matching

In the procedure matching stage, we use a combination of name-based and hashing-based methods.

The first idea we use 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*[2]. 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 BMAT, 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.

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 hashing-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.

We also perform hashing-based pair-wise comparison between procedures that cannot be matched by name. In a bottom-up fashion, we calculate a single hash value for each unmatched procedure based on the hash values of its code blocks. The calculation is sensitive to the order of blocks within a procedure. We then compare the procedures' hash values to look for matches. As 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 substantially. To catch the latter cases, we perform the above-mentioned block trial-matching between all unmatched procedures. Unlike hashing-based pair-wise comparison, which reduces each procedure to a single hash value, pair-wise block trial-matching can find a match even when code blocks have been added to or deleted from a procedure. However, we have observed that for many programs, this method can be time-consuming and yet find very few procedure matches.

In the current version of BMAT, the above four methods are performed in the following order: matching by identical names, hashing-based pair-wise comparison, matching by similar names, and pair-wise block trial-matching. We put matching by similar names after hashing-based pair-wise comparison because the former has a high error rate for some modern applications where many procedures have similar names.

2.2. Data Block Matching

As with code block matching, we use a hashing-based algorithm to match data blocks in the two binaries. Relocation entries pose challenges because their values tend to change from build to build. For our implementation, we chose to exclude relocation entries from the hashing, although more sophisticated methods are possible. For data blocks that are not matched by the hashing-based algorithm, we try to match them according to their positions in the program. If a pair of

[2] 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*.

unmatched data blocks are sandwiched by two pairs of matched data blocks in the program, we make them a match as long as their sizes are not too different.

We match the data blocks before the code blocks in order to use data matching results to help code matching. For example, if two instructions refer to two data blocks that are already matched to each other, we can consider the two instructions a match even if the data addresses are different. On the other hand, it is also justifiable to match the data blocks after the code blocks. In that case, the code matching results can help us analyze the relocation entries and improve data matching accuracy. After trying out both approaches on several programs, we chose to perform data matching first because it is simpler and often provides critical clues for 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 data matching with code matching.

2.3. Basic Block Matching

Within each pair of matched procedures, we use a hashing-based 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 3.

The hashing-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 remain 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 hashing-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.

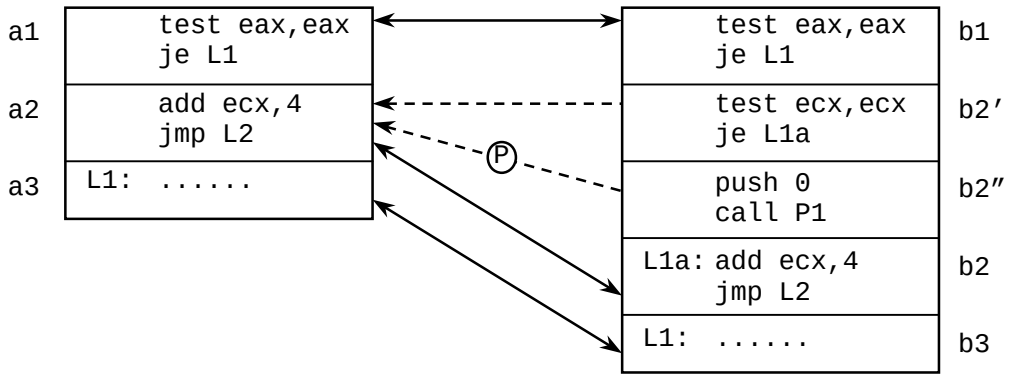

Figure 3. Example of Control-flow-based Basic Block Matching

In this example, two new blocks (`b2′` and `b2″`) are added to the fall through path of a conditional branch. During control-flow-based block matching, these two blocks are matched to `a2`, the fall through block of the matching branch in the old version. Note that

b2, b2′ and b2″ are matched to the same block. However, each new block is still matched to only one old block, so there is no ambiguity when propagating profile information.

A "partial" flag is associated with the match from block b2″ to block a2, 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.

3. Hashing-and-Matching Algorithm Details

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

3.1. Types of Code Changes

In order to match as many code 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.
- *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:

1. The source instruction is modified to refer to a different target block. This is a direct change to the block address operand.
2. 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.
3. 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.

Due to 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 offset of the target block from the beginning of the procedure and its offset 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 the 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 3.2.

- *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.
- *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.

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.

3.2. 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 1 gives an overview of all fuzziness levels, followed by detailed description for each level.

Fuzziness Level	Numerical Address Offset	Register Allocation	Block Address Operand	Operand	Opcode
0	all	all	• target block's match • target block's extended name • target procedure name or branch offset within procedure • target block's distance from beginning of procedure	all	all
1	none	EAX/ECX/EDX: dependency only	same as level 0	all	all
1a	same as level 1, but includes only the last instruction in each block				
2	none	EAX/ECX/EDX: dependency only	• target block's match • target block's extended name • target procedure name or branch offset within procedure	all	all
3	none	EAX=ECX=EDX EBX=EDI=ESI	target procedure name or branch direction within procedure	no immediate none for *return*	all
3a	same as level 3, but includes only the last instruction in each block				
4	none	N/A	N/A	type only	all
5	none	N/A	N/A	none	group

Table 1. Matching Fuzziness Levels

▪ 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 always a correct match. This is demonstrated by the example in Figure 1. In our current implementation, level 0 is not used in hashing-based 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. However, we retain some information on the use of these three registers by including their dependencies within each basic block in the hashing. These three registers are allocated for arithmetic calculations by the compiler that generated our benchmark programs.

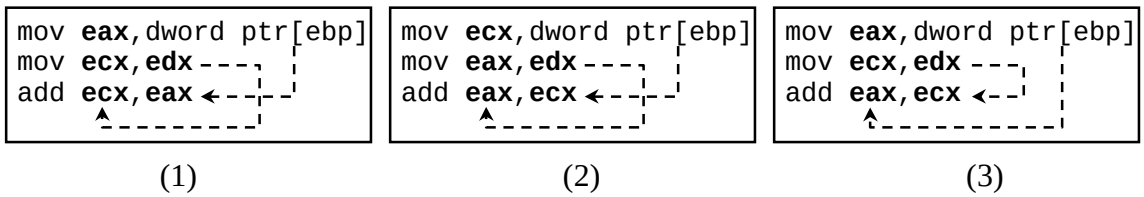

Figure 4. Excluding Register Allocation and Including Register Dependency

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 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.
- Level 3. At this level:
  - All immediate operands and operands of *return* instructions 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, the matching status and extended name of target blocks are no longer used. The 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, i.e., 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 the types (but not the contents) of its 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.

3.3. 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. We break the ties simply by matching the candidates in their order of appearance in the program. At the higher fuzziness levels, we also use block trial-matching (see Section 2.1) to reduce the rate of 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 seldom changes 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 5). 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.

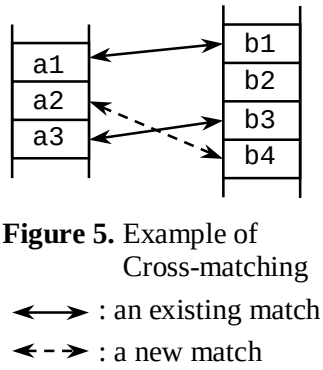

Figure 5. Example of Cross-matching

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.

3.4. 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, we perform six passes during hashing-based procedure matching. They use 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.

3.5. Example

To demonstrate our block matching algorithm, Figure 6 shows the matching results for a procedure from a common Windows NT application.

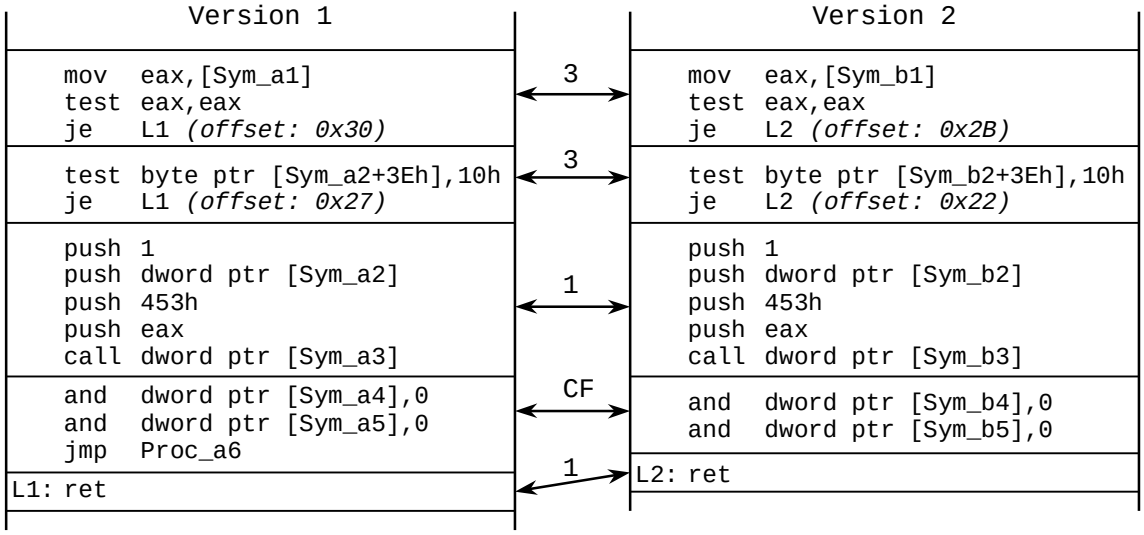

Figure 6. An Example of Code Block Matching

The number on each arrow is the annotation that indicates the fuzziness level at which the match is made. CF = control-flow-based.

In this case, 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. 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 we used for testing.

4.1. Running Time

Table 2 lists the running time of BMAT for several 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 on the matching algorithm.

Weeks Apart	File Name	File Size (KB)	Running Time (seconds)			Speed (Minute / MB)
			Build	Match	Total	
3	dhcpcsvc.dll	122	2	2	4	0.6
3	netlogon.dll	477	7	7	14	0.5
3	browseui.dll	1080	15	18	33	0.5
3	shell32.dll	3360	32	84	116	0.6
21	mshtml.dll	5628	60	178	238	0.7

Table 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

4.2. Matching Rate

Table 3 lists the matching rate results for `mshtml.dll`. Though not necessarily proving the accuracy of our algorithm, these results demonstrate how matching rate decreases when the builds are longer apart in time. It also shows that `mshtml.dll` did not change drastically over several months of time, a positive sign for the potential of stale profile propagation. It is worth noting that the rate of program change varies for different applications and different phases of product development. Even within the same project, some program modules are modified more often or more significantly than the others. Late in the development and testing process, there are usually only small changes.

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

Table 3. Matching Rates for mshtml.dll (6 Versions)

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.

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 and Table 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.

File Name	Build Dates and Number of Procedures	Number and Percentage of Matched Procedures* After			
		matching by identical names	hashing-based comparison	matching by similar names	block trial-matching
mshtml.dll	older: 03/15/99, 12891	12235	12838	12839	12839
	newer: 08/11/99, 12966	94.36%	99.01%	99.02%	99.02%
shell32.dll	older: 05/11/99, 7473	7098	7191	7222	7224
	newer: 06/15/99, 7685	92.36%	93.57%	93.98%	94.00%

* : in the newer build

Table 4. Procedure Matching Rates at Different Stages

Table 4 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 hashing-based method noticeably improves the results. As mentioned in Section 3.1, 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 BMAT.

File Name	Build Dates and Number of Code Blocks	Number & Percentage of Matched Code Blocks* After		
		hashing (first pass only)	hashing (all passes)	hashing and control-flow-based
mshtml.dll	older: 03/15/99, 155996	143643	155098	156638
	newer: 08/11/99, 157417	91.25%	98.53%	99.51%
shell32.dll	older: 05/11/99, 88906	73761	84189	87694
	newer: 06/15/99, 90538	81.47%	92.99%	96.86%

* : in the newer build

Table 5. Code Block Matching Rates at Different Stages

Table 5 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. 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[3]. 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 *fresh* profile directly collected for the newer version. As far as we know, there are no well-established metrics for evaluating propagated profile information. In this paper we propose two metrics:

Branch Prediction (B.P.) metric: We perform static branch prediction on the newer version using both the propagated profile[4] and the fresh profile, and calculate the dynamic success rate of each prediction. The Branch Prediction metric is the ratio of the two success rates, i.e., the success rate using the propagated profile divided by the success rate using the fresh profile. A B.P. metric of 100% indicates that the propagated profile is as effective in branch prediction as the fresh profile. This metric measures how well BMAT works on the most frequently executed part of the program.

[3] 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 an incompatibility problem. The automated test exercises about 27% of the code blocks in mshtml.dll.

[4] If a branch in the newer version is not covered by the propagated profile, we predict the branch to be "taken."

Code Coverage (C.C.) metric: For either profile, we assign each code block in the newer version a *coverage bit* of 1 or 0 based on whether it has a non-zero execution count in the profile[5]. A block is classified as "agreed" if its coverage bit is the same according to the propagated profile or the fresh profile. The Code Coverage metric is the percentage of "agreed" code blocks among all code blocks in the newer version. A C.C. metric of 100% indicates that the propagated profile accurately describes which blocks in the newer version will be executed. 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 "fresh profile."

Results from both profile propagation and self-comparison are given in Table 6. In these experiments, error rate from profile propagation is under 0.13% for branch prediction, about 16 times the largest execution uncertainly 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 uncertainly factor. These results show that it is feasible to use propagated stale profiles on newer versions. Assuming 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.

Build Date		03/15/99	04/16/99	07/15/99	07/27/99	08/11/99
Matched against self	B.P.	99.997%	99.997%	99.997%	99.992%	99.996%
	C.C.	99.94%	99.95%	99.96%	99.83%	99.87%
Matched against 08/11/99	B.P.	99.876%	99.876%	99.894%	99.998%	--
	C.C.	99.59%	99.61%	99.55%	99.93%	--

Table 6. Profile Propagation and Self-Comparison Results for mshtml.dll (5 Versions)

B.P.: Static branch prediction success rate using the propagated profile (BMAT-converted profile from the older version) divided by the rate using the fresh profile (profile collected directly for the newer version).

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

[5] If a code block in the newer version is not covered by the propagated profile, its coverage bit is 0.

Table 7 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 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 nonetheless promising. Throughout the table, 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).

Name		browseui	comctl32	explorer	netlogon	shdocvw	shell32	shlwapi	webcheck	Average
M.R.	A-D	99.71%	99.66%	97.80%	100.00%	99.62%	96.86%	99.45%	98.27%	98.92%
	B-D	99.96%	99.92%	99.03%	100.00%	99.67%	99.40%	99.82%	99.30%	99.64%
	C-D	99.98%	99.99%	99.03%	100.00%	99.85%	99.97%	99.91%	99.33%	99.76%
B.P.	A-D	95.94%	99.70%	99.97%	99.64%	97.81%	97.76%	99.61%	100.00%	98.80%
	B-D	96.13%	99.74%	100.00%	99.84%	98.47%	99.19%	99.82%	99.07%	99.03%
	C-D	96.51%	99.90%	100.00%	99.83%	98.50%	99.24%	99.70%	99.07%	99.09%
C.C.	A-D	98.53%	97.52%	93.82%	98.17%	99.58%	96.67%	97.23%	100.00%	97.69%
	B-D	98.43%	98.18%	97.70%	99.51%	99.65%	98.83%	98.79%	99.98%	98.88%
	C-D	98.73%	98.58%	99.65%	99.50%	99.65%	98.85%	99.08%	99.98%	99.25%

Table 7. Matching Rates and Profile Propagation Results for Eight Windows 2000 DLLs (4 Versions)

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 (code block). Percentage of successfully matched code blocks in the newer build (D)

B.P., C.C.: Branch Prediction and Code Coverage metrics for profile propagation (see Table 6 caption)

5. Related Work

Most existing PBO systems do not address the issue of stale profile data. In these systems, the profiling process starts over after any program transformation. The only system we are aware of that employs binary matching for re-using stale profiles is Spike, an optimization system for Alpha executables (Cohn et al. 1997). In Spike, profiles collected for one version continue to be used for successive builds if the changes are small. However, Spike only conducts matching on

the procedure level, and discards profile information for any procedure whose control flow graph has changed. With BMAT, profile information is propagated on basic block level and for as many blocks as possible, even in procedures that have partially changed. Spike's binary matcher does not use a procedure's name or most of its code contents. It generates a control flow graph signature for each procedure, and uses a minimum edit distance algorithm to find matches between the signatures.

Some other systems deal with stale profiles without using binary matching. Morph (Zhang et al. 1997), a system for automatic and continuous profiling and optimization, addresses the issue of stale profiles caused by 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 produces a mapping between the old and new intermediate representations so that stale profiles can be converted for use on the new version. These requirements on the intermediate representation and the optimizer are difficult to fulfill on widely available commercial systems. More importantly, 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 different approach to the problem of stale profiles, as profile information attached to source code can be carried along when the source is modified. BMAT is designed for a different type of situation, i.e., binary optimization without access to program source or additional information from the compiler.

There has been a lot of work on comparing two versions of a binary program or two different binary programs for reasons other than stale profile propagation. The most common purpose for binary comparison is efficient code patching (Baker et al. 1999; Coppieters, 1995). Code patching algorithms generally look for exact or almost-exact matches and deal with a fixed set of simple changes. The goal of stale profile propagation is much broader. In order to match as much of the program as possible, including substantially altered sections and entirely new sections, our algorithm is designed to accommodate all types of changes from the procedure level down to the opcode/operand level. A project by Baker and Mander (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 on matching them on the code and data block level. Debray et al. (1999) compared a binary with itself to find redundancies for the purpose of code compression. Instead of trying to find only identical instruction sequences, they detected "similar" basic blocks based on instruction opcodes, then looked for transformations that make them identical. For their purpose, the locations of the basic blocks do not matter to the matching process. For BMAT, block locations are extremely important to the correctness of matching, thus requiring different strategies when looking for matches.

The above binary comparison projects and our work contain some similar ideas on looking beyond small changes to find similar codes. However, these projects deal with code blocks that are identical or will become identical after simple transformation, with the goal of reducing overall code size. For stale profile propagation, we match the blocks based on both block contents and program structure, with the ultimate goal of obtaining a propagated profile that accurately describes the newer version.

6. Summary

For the purpose of stale profile propagation, we have designed BMAT, a tool for finding matches between code and data blocks in two versions of a binary program. Our implementation uses a hashing-based algorithm and a series of heuristic methods to find correct matches for as many program blocks as possible. The algorithm first matches procedures, then basic blocks within each procedure. Multiple passes of matching are performed with varying degrees of fuzziness. This process allows good matches to be found even with shifted addresses, different register allocation, and small program modifications. For program builds that are weeks or even months apart, BMAT can often find matches for over 99% of code blocks. BMAT is fairly fast. On a Pentium II 200MHz machine, the total running time ranges from four seconds for a 122KB program to under four minutes for a 5.5MB program.

A more important goal of BMAT is that the propagated profile accurately describes how the current program version will behave. In this, our results are promising. Across a set of large DLLs from Microsoft Windows 2000, propagated information from profiles six weeks old can be used to predict branch biases with a success rate almost 99% as good as that using freshly collected profiles. For identifying which blocks will be executed, the correctness rate is close to 98%. Propagated information from profiles that are closer in time performs even better on these two measures.

In our future work, we hope to demonstrate and quantify performance gains of profile-based optimizations from doing stale profile propagation. Profile propagation also introduces new issues in profile management, such as how to combine propagated profiles with fresh profiles collected for the new build. These issues present opportunities for future research. Finally, we would like to explore other uses of binary matching in program testing and patching.

References

Albert, E. (1999). A Transparent Method for Correlating Profiles with Source Programs. In *Proceedings of the Second ACM Workshop on Feedback-Directed Optimization*.

Baker, B. S., & Manber, U. (1998). Deducing Similarities in Java Sources from Bytecodes. In *Proceedings of the 1998 USENIX Technical Conference*.

Baker, B. S., Manber, U., & Muth, R. (1999). Compressing Differences of Executable Code. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Compiler Support for System Software*.

Cohn, R. S., Goodwin, D. W., & Lowney, P. G. (1997). Optimizing Alpha Executables on Windows NT with Spike. *Digital Technical Journal*, Vol. 9, No. 4.

Coppieters, K. (1995). A Cross-platform Binary Diff. *Dr. Dobb's Journal*.

Debray, S., Evans, W., & Muth, R. (1999). Compiler Techniques for Code Compression. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Compiler Support for System Software*.

Srivastava, A., et al. (1999). Vulcan. Tech. rep. TR-99-76, Microsoft Research.

Zhang, X., Wang, Z., Gloy, N., Chen, J. B., & Smith, M. D. (1997). System Support for Automatic Profiling and Optimization. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*.