Module-aware Translation for Real-life Desktop Applications

Jianhui Li
Software and Solutions Group
Intel Corporation
86-21-52574545-1654
Jian.hui.li@intel.com

Peng Zhang
Software and Solutions Group
Intel Corporation
86-21-52574545-1203
Jeremy.zhang@intel.com

Orna Etzion
Software and Solutions Group
Intel Corporation
972-4-565-5720
Orna.etzion@intel.com

ABSTRACT
A dynamic binary translator is a just-in-time compiler that translates source architecture binaries into target architecture binaries on the fly. It enables the fast running of the source architecture binaries on the target architecture. Traditional dynamic binary translators invalidate their translations when a module is unloaded, so later re-loading of the same module will lead to a full retranslation. Moreover, most of the loading and unloading are performed on a few "hot" modules, which causes the dynamic binary translator to spend a significant amount of time on repeatedly translating these "hot" modules. Furthermore, the retranslation may lead to excessive memory consumption if the code pages containing the translated codes that have been invalidated are not timely recycled. In addition, we observed that the overhead for translating real-life desktop applications is a big challenge to the overall performance of the applications, and our detailed analysis proved that real-life desktop applications dynamically load and unload modules much more frequently as compared to popular benchmarks, such as SPEC CPU2000. To address these issues, we propose a translation reuse engine that uses a novel verification method and a module-aware memory management mechanism. The proposed approach was fully implemented in IA-32 Execution Layer (IA-32 EL) [1], a commercial dynamic binary translator that enables the execution of IA-32 applications on Intel® Itanium® processor family. Collected results show that the module-aware translation improves the performance of Adobe* Illustrator by 14.09% and Microsoft* Publisher by 9.73%. The overhead brought by the translation reuse engine accounts for no more than 0.2% of execution time.

Categories and Subject Descriptors
D.3.4 [**Processors**]: *Compilers, Optimization*

General Terms: Algorithms, Design, Performance

Keywords: Dynamic binary translation, dynamic loaded module, translation reuse, memory management

1. INTRODUCTION
Dynamic binary translation offers solutions for transparently running existing applications of source architecture on a new architecture without recompiling the source code [2] [3] [4]. The dynamic binary translator takes the control when the user launches a source architecture application on the target architecture. It decodes the source binary, translates each instruction of the source architecture into instruction(s) of the target architecture, optimizes the instructions into the final translated binary, and then executes the translated binary to simulate the application. The translation is often performed on demand, and usually on a block-by-block basis. It will be triggered again when the execution reaches the code that has not been translated before. To improve the performance of dynamic binary translators, researchers in this field typically study relatively small applications [5] [6] [7], such as the SPEC benchmarks [8]. For most SPEC benchmarks, the computation are dominated by several hot "spots", which indicates that these studies usually focus on improving the performance of the translated code block of the hottest blocks.

Efficiently running real-life applications is the key to ensure the adoption of a commercial dynamic binary translator. As most users tend to port server applications to achieve best performance, we expect that most of applications running on dynamic binary translators are desktop applications. However, our research proved that the study based on SPEC benchmarks cannot lead to practical optimizations that can be applied to porting real-life desktop applications. First of all, the translation overhead for executing real-life desktop applications on a different architecture impacts the performance more remarkably than executing the SPEC programs, because the execution requires translating a lot more instructions, which consumes considerable time and resource. Secondly, focusing only on improving the translated code can be insufficient, because real-life desktop applications spend less time in the translated code since they invoke OS system calls frequently to access I/O devices, which usually execute native code directly. When running Sysmark* 2000 [9] with IA-32 EL, only 61% of the total time is spent on the translated code, while the percentage is 98% for SPEC CPU2000 [1]. Compared with the relatively heavy workload of the Sysmark* applications, real-life desktop applications spend much less time on the translated code. Different from other researchers, we will focus on exploiting optimization opportunities in translating real-life desktop applications in this paper.

Our study shows that dynamic module loading and unloading, the typical behavior of many desktop applications, seriously impacts the translation time, memory consumption, and profiling information collection. As modular design is popularly used for building desktop applications, typically a few modules will be frequently loaded and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
VEE'05, June 11–12, 2005, Chicago, Illinois, USA.
Copyright 2005 ACM 1-59593-047-7/05/0006...$5.00.

* Other names and brands may be claimed as the property of others

unloaded during the application execution. We call these modules hot modules. In the traditional binary translators that we have previously discussed [1], the translation of a module is invalidated upon unloading and the same piece of code is retranslated when the module is loaded again for execution. Since some common functions, like DLLMain in windows DLL, are certain to be called upon module loading and unloading, and programmers usually call a small number of functions in the module, repeatedly loading and unloading the same hot module results in repetitively translating these functions, which consumes excessive time and resource undesirably. Furthermore, a memory management mechanism that is unaware of module translation may keep the translated code of the hot modules and the other modules in one code page, which leads to inefficient memory management--If the translations of the hot modules are invalidated upon module unloading, the memory may contain lots of internal fragments afterwards; even if the translations of the hot modules can be reused, the frequently reused hot module translations may be accidentally collected as garbage because the surrounding translations are out of date. Finally, since the profiling information is also lost when the translated codes are invalidated upon unloading, and the translated codes that are newly generated are stored in a different memory location, the profiling information is inaccurate and hard to be collected. For example, the hot spot in a hot module that is frequently loaded and unloaded may not be recognized, because its profiling information is lost after each unloading – the binary translator is unable to realize that the binaries in the module are translated and executed frequently.

To address these issues, we propose to add two components into the binary translator: a module translation reuse engine and a module-aware memory management mechanism. First, the translation reuse engine reserves and reuses the translated code blocks or pages of the module. To ensure the correctness of the reuse, the engine needs to verify the consistency of the binaries in the reloaded module. Simply checking the modification date of the file containing the module cannot ensure the consistency, because it's possible to keep the date unaltered before and after changing the content of the file. Without the special support from the OS, a possible way of consistency verification is to save and compare all the binaries of the module. However, this is very space and time consuming, and even impossible if some pages of the module are prohibited to be read. This paper proposes a verification method which is characterized by 100% correctness, high speed, compactness, and high quality of the translated code. Second, the module-aware memory management mechanism divides the memory resource into two categories: module-private code pages and general code pages. Each hot module has its corresponding private code page pool, and the rest of the modules share a general code page pool. With this categorization, the memory manager can better support the translation reuse, and deploy more efficient garbage collection policies for different pools.

The contributions of this paper include:

1. A powerful reuse engine that avoids repetitive translation of hot modules

2. A verification method that features 100% correctness, high speed, compactness, and high quality of the translated code

3. A module-aware memory management mechanism specially tailored for hot module translation

We incorporate the module-aware translation in IA-32 Execution Layer (IA-32EL) [1], a commercial dynamic binary translator that enables execution of IA-32 applications on Intel® Itanium® processor family. Real-life desktop applications of various types are selected to compose our benchmark suit and representative workloads are built for them. Our results show that the performance can be improved up to 14.09% for Adobe* Illustrator and 9.73% for Microsoft* Publisher, which frequently reload hot modules. The translation time drops by 28.85% in Microsoft* Publisher and 28.71% in Adobe* Illustrator, and the memory consumption drops by 59.46% and 24.04% respectively. The overhead brought by the translation reuse engine is almost ignorable, which merely accounts for less than 0.2% of the translation time.

The rest of the paper is organized as follows: We'll describe the general architecture of dynamic binary translators in section 2, followed by section 3, in which the characteristic and the performance impact of dynamic loading and unloading are analyzed. In section 4, we'll introduce the module translation reuse engine and illustrate the verification method used in the reuse engine. Then we'll describe the module-aware memory management mechanism in section 5. We'll present our performance data in section 6 and discuss the related work in section 7. Finally we'll summarize the paper in section 8.

2. GENERAL STRUCTURE OF DYNAMIC BINARY TRANSLATORS

Generally, a dynamic binary translation is composed of two stages: translation stage and execution stage [1] [10] [11]. At the translation stage, the binary translator reads the source binaries and translates them into target binaries. At the execution stage, the binary translator branches to execute the translated target binaries. Correspondingly, there are 2 vital components in a dynamic binary translator: an execution engine and a translation engine.

- The execution engine directs control flow through the translated blocks. When the control flow reaches a source architecture instruction that doesn't correspond to a valid translation or the translation of which is performance critical and thus is worth a retranslation for further optimization, the translation engine is triggered.

- The translation engine translates the source architecture binaries into target architecture binaries (translated code blocks) in an adaptive way. Typically there are 3 phases: the interpretation phase, the fast translation phase, and the optimization phase. When an instruction is executed for the first time, it is interpreted to simulate the instruction on the target architecture. In the interpretation phase, no translation is saved and the instruction needs to be interpreted again if it is executed for the second time. As the instruction is executed more times, the translation engine transits to the fast translation phase. In this phase, the translation engine explores source fragments from the given instruction pointer, forms basic blocks, and constructs a control flow graph. Thereafter, a fast translation method, which only adopts light-weighted optimization, is used to generate the target code blocks. The translated target code blocks are saved in the memory, and can be reused when the control is transferred to the first instruction of the block. The translated codes that are performance critical are known as hot spots, which are discovered by the profiler. These codes are retranslated in the optimization phase. The optimization phase uses most optimization techniques used in static compilers, and can be very sophisticated and time consuming. Based on the profiling information, different optimizations are adopted in generating the translated code, so that

the translation time and the quality of translated code are balanced. There are various approaches to collect profiling information, such as inserting instrumentation code into the translated or interpreted code[1] [12], using low overhead dynamic profiling tools[13], etc. The implementation of a dynamic binary translator may not contain all the phases mentioned above. However, it is critical to have both the fast translation phase and the optimization phase, so that the translator can generate optimal translation for the hot spots and translate the "cold" source binaries swiftly.

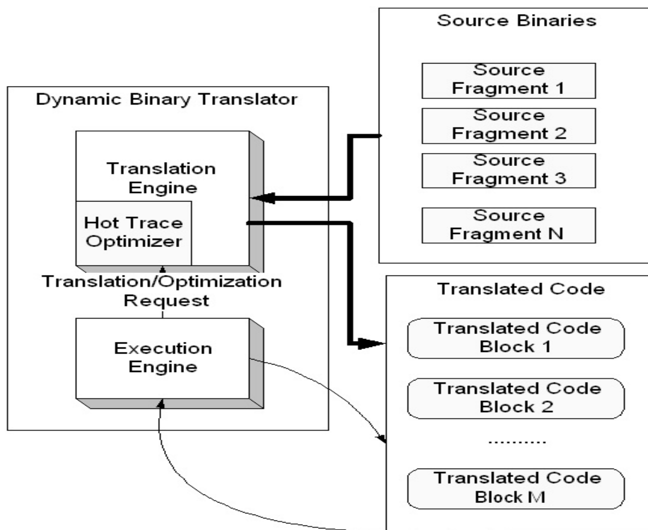


Figure 1. The General Architecture of Dynamic Binary Translators

Typically, dynamic translators use a low overhead memory management mechanism to exploit temporal locality by attempting to keep useful, active translations in the code pages [1] [10] . As shown in Figure 2, the translated code blocks are saved in the code pages that are organized into two pools: a used page pool and a free page pool. Once a new translated code block is generated, it is added to the latest allocated code page in the used page pool. If there is no space in the page to hold the block, the memory allocator is invoked to provide a new code page. The memory allocator first tries to allocate a page from the free page pool. If there is no page in the free page pool, it requests memory resource from the system. The allocated page is added to the used page pool. If the pages in the used page pool exceed a threshold, the garbage collector is triggered to recycle code pages by moving them from the used page pool to the free page pool. The pages in the free page pool can be freed and returned to the system if there are too many free pages. The arrows in the Figure 2 show the page flow among these two pools and the memory provided by the system. With very low overhead, the garbage collector tries to identify a selection of translated blocks that are not likely to be executed in the near future. A commonly used policy is Least-Recently Created (LRC), which marks each translated code page with its age, and recycles the old pages in the same order as they were created [14].

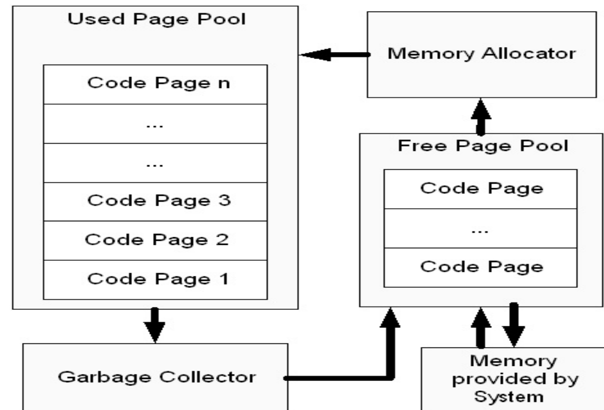


Figure 2. Memory Management Mechanism

3. THE CHARACTERISTICS OF DYNAMICALLY LOADING MODULES AND ITS PERFORMNACE IMPACT

In the characteristic analysis， 4 real-life desktop applications are studied, namely Microsoft* Publisher 2000, Adobe* Illustrator 9.0.1, Acrobat* Reader 4.0, and Macromedia* Dreamweaver MX, covering the fields of desktop publishing, graphic design, document reading, and Web page editing. The 4 applications are from 3 well-known companies and are assumed to represent different programming styles. Their workload is carefully designed to reflect real users' typical operations and to prevent repetitive operations from misleading the analysis result (Details of the workload are described in section 6.)

Table 1. The Number of DLL Loading and Unloading

Application	Number of Loading	Number of Unloading
Microsoft* Publisher 2000	3684	3645
Adobe* Illustrator 9.0.1	1010	966
Acrobat* Reader 4.0	142	92
Macromedia* Dreamweaver MX	180	115
CPU2000 INT Programs	9 (average)	0
CPU2000 FP Programs	8 (average)	0

The number of DLL loading and unloading performed during the execution of the 4 real-life desktop applications and SPEC CPU2000 programs are listed in Table 1. (The number of loading and unloading can be unequal, because some modules are not unloaded until the program terminates.) It is obvious that Microsoft* Publisher 2000 and Adobe* Illustrator 9.0.1 load and unload DLLs a lot, while Acrobat* Reader 4.0 and Macromedia* Dreamweaver MX perform less such operations. The SPEC programs load only a few DLLs and don't unload them at all.

Table 2. Hot module

Application	Number of Loadings	Number of Hot Modules	Number of Loadings performed for Hot Modules	Number of Loadings performed for the Hottest Module
Microsoft* Publisher	3684	121	3515	323
Adobe* Illustrator	1010	148	649	125
Acrobat* Reader	142	10	42	11
Macromedia* Dreamweaver	180	14	61	11

For real-life desktop applications, a number of DLL loadings and unloadings are caused by repeatedly loading and unloading hot modules, where we define the hot module as the modules that have been loaded for more than twice in a program's execution,. According to Table 2, on average, each hot module has been loaded for 22 times during the execution of Publisher and Illustrator.

Loading and unloading DLLs frequently can amplify the translation overhead, including the execution time spent on the translation and the memory consumed by the translated code. The translation overhead is a notable performance issue to real-life desktop applications though it's not so important to CPU2000 programs. In Figure 3, we listed the translation time of the 4 real-life desktop applications and the CPU2000 programs, running on an IA-32 Execution Layer version without the translation reuse mechanism and module-aware memory management, and in Figure 4, the memory consumption are listed correspondingly. From these figures, we can see that the translation overhead is ignorable for the performance of CPU2000 programs: Less than 1% of the execution time is spent on the translation and the translated codes occupy only 2.66 MB. But the translation overhead is much higher in the 4 real-life desktop applications: On average, 12.2% of the execution time is spent on the translation and 98.21 MB are consumed by the translated code. In addition, the data also indicates that the translation overhead of Microsoft* Publisher and Adobe* Illustrator, which load and unload DLLs frequently, is higher than the other two, which load and unload DLLs less frequently. In the applications that load and unload DLLs a lot, hot modules consume a considerable part of the translation time and memory, as shown in Figure 5 and Figure 6.

Frequently loading and unloading DLLs may result in redundant translations. Since some functions are always called upon DLL loading, like DLLMain, and some programmers have the habit of unloading a DLL immediately after calling functions in it and reloading the DLL when those functions are needed again, a possible approach to decrease the translation overhead is reusing DLL translations rather than simply discarding them when the DLL is unloaded and regenerating them when the DLL is reloaded. The approach can speed up programs that suffer from repetitively translating functions caused by frequently loading and unloading DLLs.

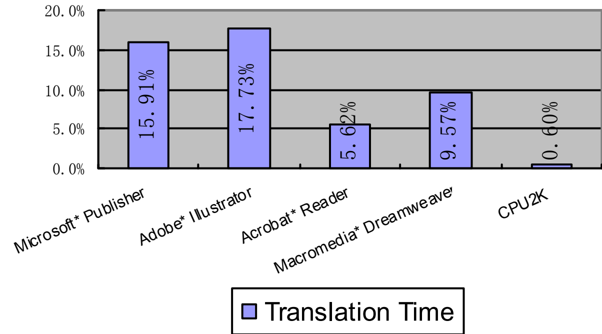


Figure 3. Translation Time

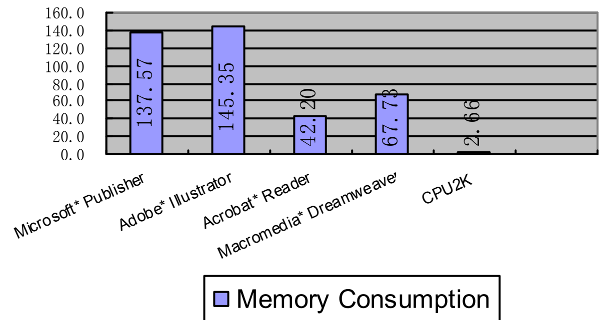


Figure 4. Memory Consumption

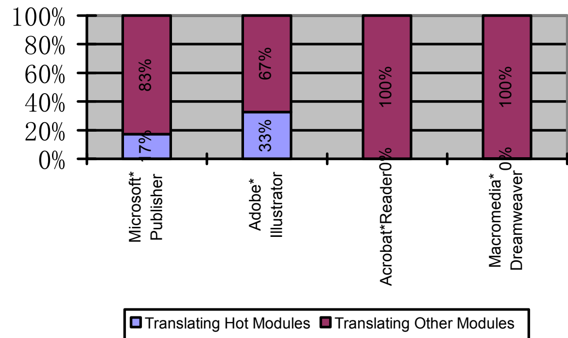


Figure 5. Translation Time for Hot Modules

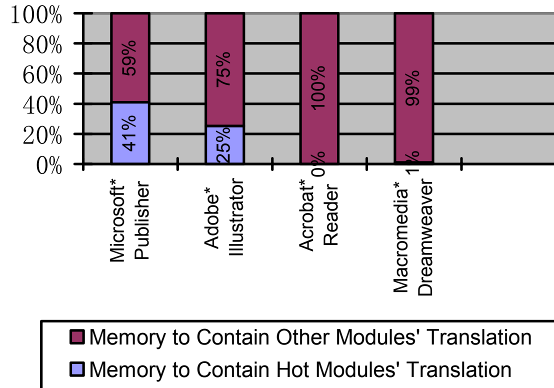


Figure 6. Memory Consumption for Hot Modules

4. MODULE TRANSLATION REUSE ENGINE

4.1 Framework

The module translation reuse engine is added to the dynamic binary translator as shown in Figure 7. Once a module is recognized as a hot module, the reuse engine saves the source binaries and their translations in a dedicated memory area. The reuse engine only saves and compares minimum amount of binaries to ensure the reusability of the preserved translations. This is described in detail in the section 4.2. Before the translation engine translates a piece of the source binary, it requests the reuse engine to check whether these binaries are translated before and the previous translations can be reused. The reuse engine first checks whether the entry address of the source binary belongs to a hot module. The module to which the entry address belongs can be easily determined by searching the module list that contains all the modules. Each module distinguishes itself by name, image size and the base address. If the current module is found to be a hot module, the reuse engine searches for its saved translations to check whether the binaries have been translated before, then compares the saved binaries to verify whether the translated code blocks can be reused.

There are three stages to accomplish the translation reuse:

1. Translation Reservation

When a translated code block is invalidated, for example, due to module unloading, it is preserved by the reuse engine and saved by the execution engine. The reuse engine does the bookkeeping about where the translations are saved, to which hot module the translations belong, and the translated code block descriptors that describes the entry addresses of the source binaries and their translated blocks.

2. Source Binaries Verification

There are two steps in this stage: save and comparison.

a) If the reuse engine decides to reuse the translation for a piece of the source binaries, such as the binaries in hot modules, it saves the source binaries after translating them successfully. The reuse engine minimizes the verification overhead by saving a minimum set of source binaries that determine the semantics of the translation.

b) Before the translation engine translates a piece of the source binary, it requests the reuse engine to check whether a preserved translation associated with the current instruction address exists, and whether the translation is reusable. The reuse engine compares the saved source binaries with their counterpart in the module image. If they're exactly the same, the reserved translations associated with the piece of binaries are declared as reusable. No new translation is needed for these binaries.

3. Translation Revivification

Before executing the reserved translation that is declared as reusable, the reuse engine is requested to revive the reserved translations. After the revivification, the reserved translation can be seen by the execution engine.

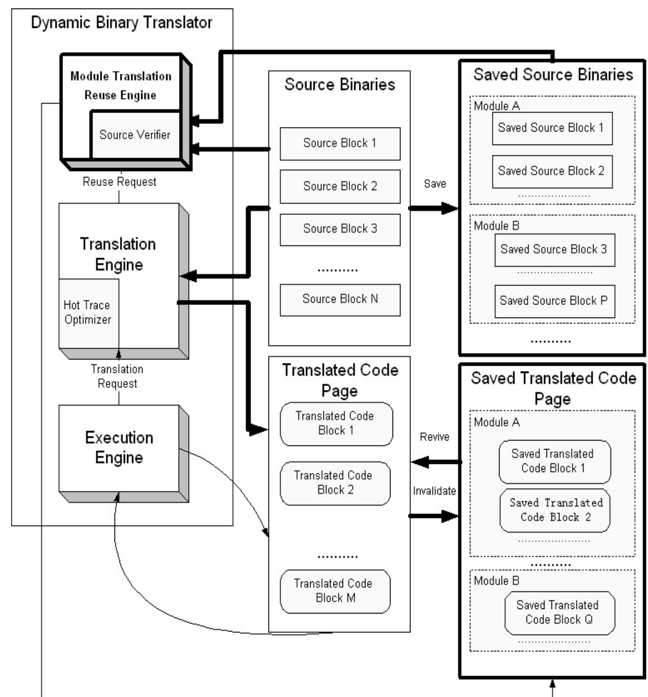


Figure 7. Framework for Module Translation Reuse Engine

4.2 Source Binaries Verification

Before reusing a piece of the translation, we must check if the source binary, which the reused translation is about to simulate, is exactly the same as the binary from which the translation was translated. If they're different, the reuse is possibly incorrect in functionality, because the behaviors of the reused code fragment may be different from the source binary it simulates. In this paper, we define checking the source binaries as source binaries

verification, which is performed by the reuse engine. The reuse engine saves the source binaries as they're translated and compares them, in bitwise, with the corresponding parts of the executable image to get the translations' reusability. For performance reasons, the verification overhead must be minimized. Apparently, the verification overhead is proportional to the amount of source binaries that are saved and compared. We don't want to save and compare the entire text section of a DLL, because this will take much time and memory, and can negate the improvements gained from the translation reuse. We don't want to use the checksums of the source binaries in the verification either, because the checksum computation is time consuming.

The proposed method in this paper decreases the overhead by saving and comparing only those source binaries that determine translations' behavior and thus affect the correctness of the translation reuse. We use Referred Source Binaries (RSB) to denote the minimum set of source binaries, which must be verified to achieve the reusability of a translated code block..

Definition: Referred Source Binaries (RSB)

For a translated code block T, if there is a set of source binaries that determines its behaviors, then this set is defined as the Referred Source Binaries of T, denoted as RSB (T). This can be implemented by following algorithm:

Input : A source fragment T and

Output: Its RSB set.

Algorithm:

```
    RSB(T) = {T};
    for each optimization O applied to T
       for each source fragment S referenced in applying O to T
            RSB(T) = RSB(T)  U {O};
       end
    end
```

For the translated code block T, if every element in RSB (T) is unchanged, reusing T would be 100% correct, even if there are changes in other parts of the source binaries in the same module. Since RSB (T) is the minimum set of source binaries that determine the semantics of T, comparing any real subset of the Referred Source Binaries cannot ensure the 100% correctness of reusing T. Therefore, for each translated code block, saving and comparing its RSB is the most efficient way to verify its reusability.

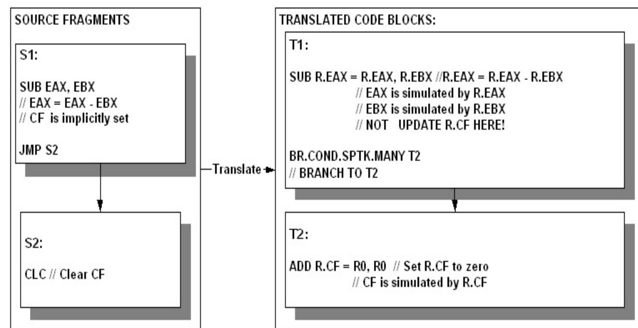


Figure 8. Example for Referred Source Binaries Affected by Global Optimization

Global optimization may refer to the information passed from some predecessors or successors of the source code block, and thus adds more binaries into the RSB set of translated code block. The predecessor and successor should also be added to the RSB set since they determine the semantics of the translated block. In the example shown in Figure 8, two source fragments (IA-32 code), S1 and S2, are translated into two translated code blocks (Itanium code), T1 and T2. The EFlag optimization uses the live information of S2 when translating S1 into T1. T1 does not update CF because it expects that CF is assigned to zero in S2 anyway. If S2 is changed to read CF instead of clearing it, S1 must be translated into a new translated code block, in which CF should be updated. If T1 is reused as S1's translation, the translated code block of S2 would get an incorrect value of CF. So, for the translated code block T1, RSB (T1) = {S1, S2}, which means that both S1 and S2 should be checked before reusing T1.

Global optimizations can lead to repetitive verification, because they may propagate the information of one source fragment to several surrounding blocks for improving their translations. Therefore, a source fragment can be in multiple translated code blocks' RSB sets. For example, in Figure 8, both RSB (T1) and RSB (T2) contain S2. If two copies of S2 are stored for T1 and T2 respectively, some memory will be wasted and some execution time would be wasted by comparing the two identical copies with the new executable image.

A simple way to avoid repetitive verification is to disable the global optimization functionality when translating the binaries for hot modules. Without global optimization, the semantics of the translated code block is fully determined by the source fragments from which it is translated. So the RSB of each translated code block only contains its direct source binaries counterpart, e.g. RSB(T1) = {S1}. However, global optimizations are crucial in improving the performance of the translated code block. For example, the expected stack top analysis in the IA-32 Execution Layer plays a crucial role in efficiently simulating the IA-32 FP stack [1]. Disabling all of them will seriously degrade the quality of the translation and is unacceptable to commercial dynamic binary translators.

Our verification method avoids repetitive verification by adopting group verification. Group verification groups the translated code blocks together. The semantics of the translated code block group is determined by Group Referred Source Binaries (GRSB).

Definition: Group Referred Source Binaries (GRSB)

For a translated code block group G, G = {T1, T2, …, Tn}, the Group Referred Source Binaries of G is the union of each group members' RSB set. This can be expressed as GRSB(G) = Union(RSB(T1), RSB(T2), …, RSB(Tn)).

When the dynamic binary translator needs to reuse the translations in the group, it verifies the GRSB set of the group with the corresponding parts in the executable image. If the binaries are consistent, then all the translated code blocks in the group are declared as reusable. If a source fragment is in two translated code blocks' RSBs and the two translated code blocks are in the same group, only one copy of the source fragment is created and the corresponding part in the executable image is compared once. In this way, the method of group verification removes the repetitive verification inside the group.

The side effect of group verification is that some translated code block's RSB elements are compared but the translated code block is not actually reused. Consider the example in Figure 8, if we put T1 and T2 in a group and the dynamic binary translator tries to reuse T2, then the translator will compare the source fragments associated with the group with their corresponding parts in the execution image, in which S1 is included. However, S1 has nothing to do with T2, and if S1 is not going to be an RSB element of an actually reused translated code block, the execution time spent in comparing S1 is wasted.

To balance the repetitive verification and useless comparison and thus achieve the minimum verification overhead, the reuse engine can limit the size of a group when the group size exceeds a predefined threshold. A simple grouping algorithm works by adding translated blocks that share elements in the RSB sets to a same group. When the group size reaches the threshold, no translated code blocks will be added to the group, so that useless comparison leading to inefficient verification can be avoided. The threshold is set to 2 mega bytes in our implementation.

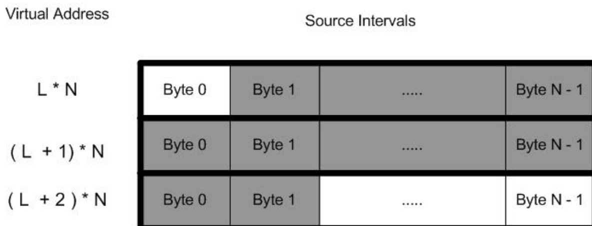
4.3 Implementation consideration

This section introduces the techniques we used to minimize the overhead of the IA-32 EL's reuse engine.

4.3.1 Using Source Interval to Simplify Memory Accesses

Saving and checking RSB elements involve a number of memory accesses. The memory accesses can be ineffective, because they might be unaligned or even cause access violation exceptions. To avoid misalignment and frequently access permission checking, we partition source binaries into source intervals, and verify the source intervals that contain the RSB elements.

A source interval is a memory block that consists of N continuous bytes and starts from an address that is a multiple of N, where N is a factor of the page size. Because all the bytes in a source interval are in the same page, we need only one request to get all of their permissions. In addition, since the starting addresses of the source intervals are multiple of N, misalignment can be avoided by setting N as a multiple of the memory access size.



In the graph, a source interval consists of N byte. The bytes in gray belong to RSB elements and the white ones do not.

Figure 9. Source Interval

The side effect of source interval is that some, though a few, bytes that does not belong to RSB elements might be saved and compared as a part of the source intervals, because not all RSB elements are N-byte-aligned and N-byte long. So the length of the source interval has been tuned in IA-32 Execution Layer to avoid too many needless bytes.

The translated code blocks are grouped by module in IA-32 Execution Layer. When a module is loaded and the OS has finished patching it, all the saved source intervals associated with the module are compared with their corresponding parts in the executable image. If all the source intervals are unchanged, translations associated with the module are considered reusable and the dynamic binary translator will try to reuse them. If any one is different, the translation is not reusable and the translations, as well as their source intervals, will be discarded.

4.3.2 Reserving Translations by Moving Translated Code Block Descriptors

We don't reserve translations by copying translated code blocks, because copying so many bytes is expensive. As an alternative, we take full advantage of the fact that the execution engine of IA-32 EL accesses translated code blocks via translated code block descriptors, which are in a hash table. So we reserve translations simply by moving their translated code block descriptors out of the hash table. Details are as follows:

- There is an alternative hash table for every hot module, which contains the translated code block descriptors of the invalidated blocks that are associated with the module. The descriptors in the alternative hash table are logically invisible to the execution engine.
- When the translations of the hot modules are invalidated, the translator does not remove the translated code blocks from the memory, instead, it moves the descriptors from the hash table used by the execution engine into the alternative hash table of the module.

In addition to avoiding copying too many memory units, another benefit of this approach is that a translated code block can be easily located at the reuse stage.

4.3.3 Deep Revivification

Reusing a translated code block is actually implemented by exposing the translation to the execution engine, which can be achieved by moving the reserved translated code block descriptors back to the hash table accessible to the execution engine. This is called translation revivification in our study. To avoid invoking the revivification procedure frequently, we revive the saved translated code blocks in a method called deep revival. Deep revival means that if the successive block in a control flow of the translated code block is unique, then the successive block will be revived as well, and this process is repeated continuously.

Figure 10 shows the number of invocations of the revivification procedure with and without deep revival. It indicates that for Microsoft* Publisher and Adobe* Illustrator, deep revival reduces the number of invocations by 14.47% on average.

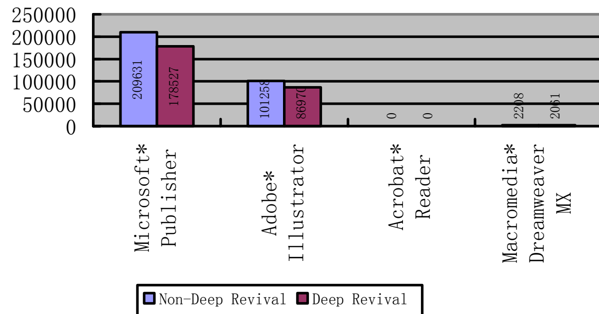


Figure 10. The Number of Invocations for Revivification

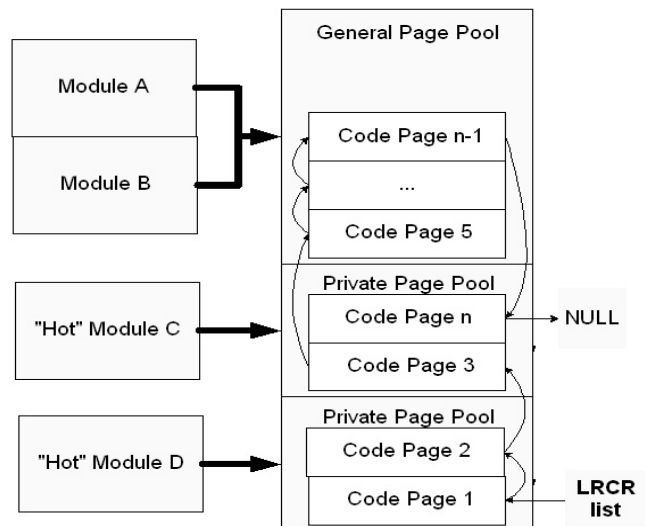


Figure 11. Module-aware Memory Management Mechanism

5. MODULE-AWARE MEMORY MANAGEMENT MECHANISM

In translators that saves the translated code blocks in the order of the translation [1], the translated code blocks of the hot modules and other modules are mixed together. Unloading a hot module without reusing its translation may lead to many internal fragments that contain invalidated translations of the module. Translation reuse can effectively avoid the internal fragments because the invalidated translations can be revived later upon module reloading. However, if the memory management mechanism is not aware of the translation revival, the translated code blocks of the hot module may be inevitably removed when other translated code blocks in the same code page meet the garbage collection criteria and the garbage collection mechanism reclaims the whole page.

The module-aware memory management mechanism organizes the translation code blocks of different modules into different pools. It divides the code page pools into two categories: module-private page pool and general page pool. Each hot module has its own corresponding private code page pool, and the other modules share the general code page pool. When a translated code block is saved, the memory management mechanism identifies the module to which the translation belongs, and then determines the page pool for the translated code block. When the hot module is unloaded, its private page pool codes are reserved for future reuse. If the translation is identified as not reusable by the verification method upon next reloading of the module, its private pool is immediately collected and recycled.

To keep the frequently revived code pages and free the pages that are not reused in the private page pool timely, the module-aware memory management mechanism uses a Least-Recently Created & Revived (LRCR) policy based on the LRC policy. Similar to the LRC policy, the LRCR policy marks each code page with its age, which is a global number that is increased each time a new code page is allocated. The LRCR policy re-marks the age of the page as if it is newly allocated when a page of a hot module is revived or, precisely speaking, when one translated block in the page is revived and becomes active. For example, in Figure 11, the first code page in the private page pool of hot module C, which is recently revived, updates its age from 4 to n, the value of the reuse counter at the time of the revival.

We implemented this by maintaining a double linked list—the LRCR list, which links all the pages from oldest to youngest. If a page is revived, we remove the page from the list, and insert it at the end of list. The garbage collector always collects pages from the beginning of the LRCR list. In this way, the code pages of the hot modules are reserved in the pool as newly created pages if they are frequently revived, and they will be moved to the beginning of the list and get recycled if the module stops to be a hot module or some pages of the hot module are not revived for a long time.

6. PERFORMANCE EVALUATION

The translation reuse and module-aware management mechanism has been implemented in IA-32 Execution Layer. All the performance measurements were done on the same system equipped with two 1.5GHz Itanium 2 processors with 6MB L3 cache. The operating system is Microsoft Windows 2003 Enterprise. Operations in the workload are typical users' operations.

6.1 Workload Description

The workload of Microsoft* Publisher is:

1. Create an Astor quick publication, accent box catalog and blank publication respectively by the wizard.
2. Drag a picture to 4 different locations.
3. Create a web site by the wizard.
4. Convert the web to print.
5. Repeat Step 3 and Step 4 for 3 times.

Without module-aware translation, the duration of the Microsoft* Publisher's workload is 90.64 seconds on top of IA-32 Execution Layer. And with module-aware translation, the duration is 81.82 seconds.

The workload of Adobe* Illustrator is:

1. Stylize a TIF image by adding arrowheads and cancel the stylization immediately.
2. Apply the effects of "Dry Brush", "Gaussian Blur", "Radial Blur", "Diffuse Glow", "Crystallize", "Mosaic

Tiles", "Gaussian Blur" and "Colored Pencil" to the image.

3. Move a JPG image up and down. Rotate it by 90%. Send it back and bring it to front.
4. Apply "Flatten Transparency" to the object and preview it.
5. Drop shadow to the image and apply the effect called "Glass" to it.

Without module-aware translation, the duration of the Adobe* Illustrator's workload is 99.63 seconds on top of IA-32 Execution Layer. And with module-aware translation, the duration is 85.59 seconds.

The workload of Acrobat* Reader is:

1. Go to chapters by bookmark
2. Search a word in a PDF document for 30 times
3. View the PDF document in different size, including full screen, actual size, fit width and etc.
4. Go to pages by thumbnails
5. Repeat Step 1 to Step 4 for 3 times

Without module-aware translation, the duration of the Acrobat* Reader's workload is 41.88 seconds on top of IA-32 Execution Layer. And with module-aware translation, the duration is 41.82 seconds.

The workload of Macromedia* Dreamweaver is:

1. Open an existing site
2. Change HTMLs in the site
3. Tutor the site

Without module-aware translation, the duration of the Macromedia* Dreamweaver's workload is 164.84 seconds on top of IA-32 Execution Layer. And with module-aware translation, the duration is 164.61 seconds.

6.2 Data and Analysis

Figure 12 shows the numbers of translated code blocks in all combination of translation reuse and module-aware memory management. It indicates that while translation reuse has a strong contribution to reducing the block numbers, the two optimizations are complementary: For Microsoft* Publisher, translation reuse, alone, reduces the block number by 58.76%; But when the module-aware memory management accompanies it, the number is reduced by 59.44%. Regarding Adobe* Illustrator, the block number declines from 542K to 411K with translation reuse and module-aware memory management, say, reduced by 24.12%. The benefit seen in Publisher and Illustrator primarily comes the fact that DLL loading and unloading no longer cause repetitive translation. Besides that, we get some bonus from avoiding re-translating the code blocks in the code pages undesirably collected by the garbage collection. When the repetitive translation caused by DLL loading and unloading as well as the garbage collection is eliminated, the memory pressure is relieved. That reacts on the garbage collection: the garbage collection is triggered less frequently and is less likely to collect translated code blocks which will actually be needed in the near future and cause redundant translation. The memory consumption, shown in Figure 13, backs up the measurement of the block numbers: The memory consumption is reduced by 59.46% in Microsoft* Publisher and by 24.02% in Adobe* Illustrator. As what we expected, neither Acrobat* Reader nor Macromedia* Dreamweaver benefits from translation reuse or module-aware memory management, because DLL loading and unloading is not done heavily in the two applications.

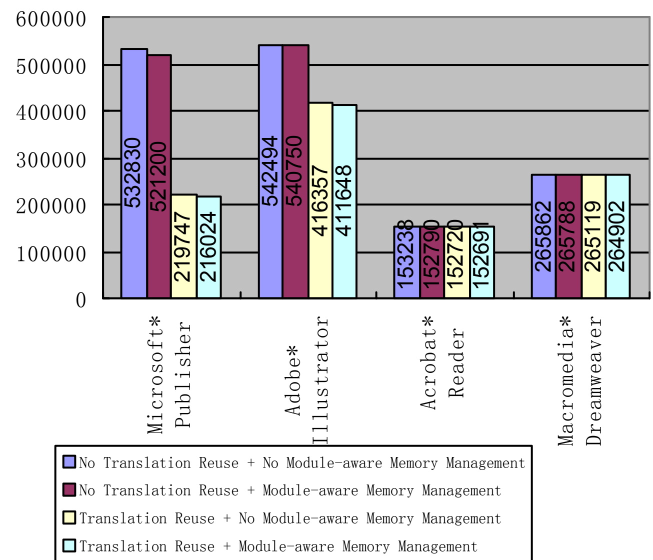


Figure 12. The Number of Translated Code Blocks

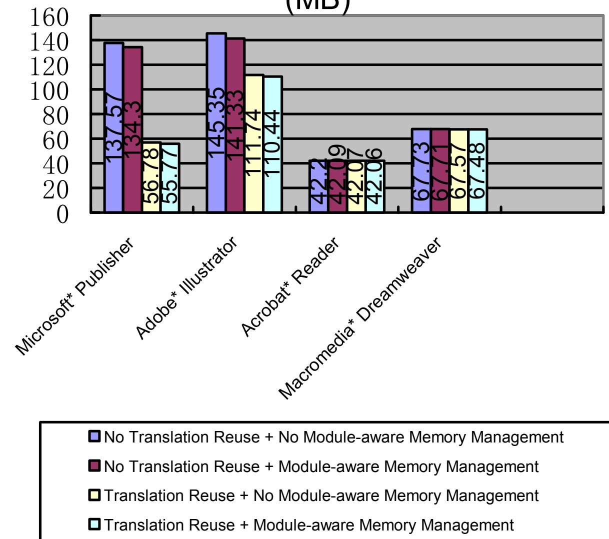


Figure 13. Memory Consumption

Figure 14 shows the execution time spent in translation. With translation reuse and module-aware memory management, the translation time drops by 28.85% in Microsoft* Publisher and 28.71% in Adobe* Illustrator. The improvement is not in direct proportion to that of block number and memory consumption, because when translation is reused, profiling gets more accurate and hot traces are possible to be different. The changes in hot traces further affect the translation time: changes can make the dynamic binary translator to select different types of optimizations to apply and the optimization overhead changes. Just as what we expected, the translation time of Acrobat* Reader and Macromedia* Dreamweaver almost keeps the same.

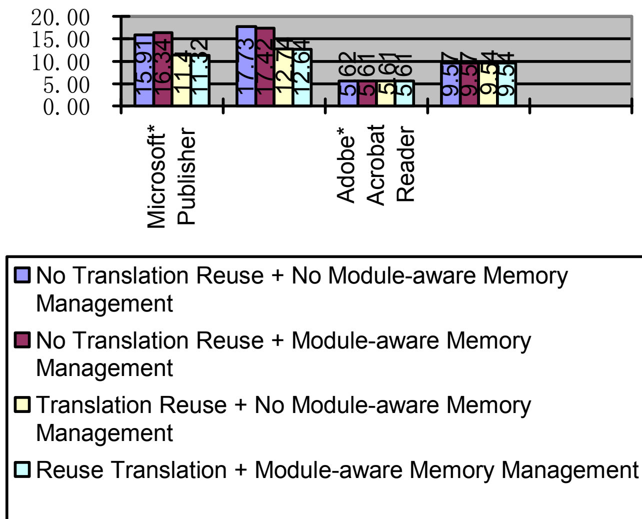


Figure 14. Translation Time

Figure 15 shows the overhead of translation reuse mechanism. For Microsoft* Publisher and Adobe* Illustrator, the overhead is no more than 0.2% of the execution time and ignorable. For Acrobat* Reader and Macromedia* Dreamweaver, which don't benefit from reusing translation, the overhead is 0%.

Figure 16 shows the speedup brought by all combination of translation reuse and module-aware memory management in the 4 applications. Applying the two brings an impressive speedup of 9.73% to Microsoft* Publisher and a speedup of 14.09% to Adobe* Illustrator. Actually, the speedup is more than what we gain from saving translation time, and that's due to when translations, as well as their profiling data, are reused, the profiling data is more accurate. That improves the quality of the translated code. For Acrobat* Reader and Macromedia* Dreamweaver, the speedup is minor and can be regarded as ignorable variance.

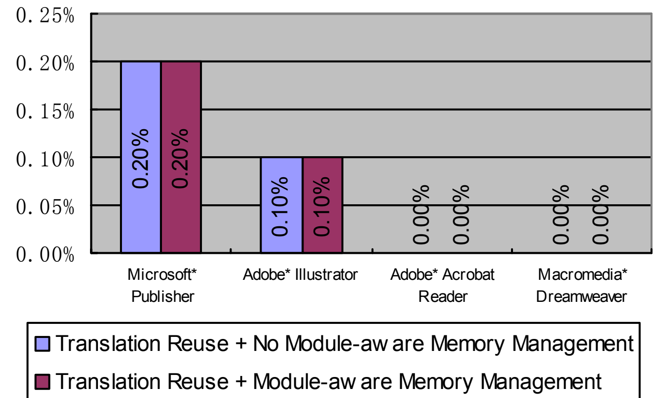


Figure 15. The overhead of Translation Reuse Mechanism

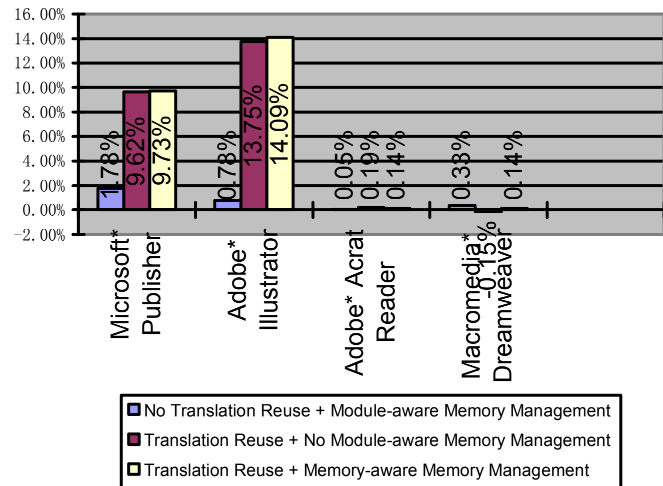


Figure 16. Overall Speedup

7. RELATED WORK

Most of the related work concerns FX!32 [12][15], a dynamic-static hybrid translator that ports applications from the IA-32 architecture to the ALPHA architecture. This translator tries to reuse translations across multiple running copies of a program. In FX!32, a background static translator creates segments of the native Alpha codes that duplicate the functionality of the x86 codes previously executed under emulation. The translated Alpha code segments will be reused next time when the IA-32 program is invoked. However, the translator does not compare the binaries of the IA-32 program to verify the reusability of the translated Alpha code segments. In

addition, FX!32, being static, saves the translations on the disk and loads it from there each time when the translations are needed, whereas our solution maintains all the translations in memory, yet achieves reusability and complete correctness.

Another interesting similarity exists in JVM [16], in which the translations of the methods of the classes that are frequently loaded and unloaded can be reserved for future reuse. However, as JVM translation is performed on the method basis and the sizes of the methods are usually much smaller than sizes of the modules, JVM can easily produce an efficient verification solution by comparing the source codes and their translations or comparing the checksum generated by them.

Most of research frameworks about binary translators only focus on speeding up the translated code of the hot spots, and typically use SPEC benchmarks as performance indicators. These researches merely address the issues in translating real-life desktop applications [5][6][7][17].

Dynamic binary translators that run under the operating system, like Transmeta CMS [10] and DAISY [18], cannot identify the module to which the binaries being translated belong, because the modules can only be seen by the operating system and the applications running on it. Other products in the field, such as HP Aries[11], did not report similar work in module translation reuse and module-aware memory management mechanism.

8. CONCLUSION

In this paper, we presented the module-aware translation, which consists of a module translation reuse engine and a module-aware memory management mechanism, for resolving the performance impacts brought by frequently loading and unloading run-time modules in real-life desktop applications. We implemented the module-aware translation component in IA-32 EL, and evaluated the performance results. Our research results showed that the performance can be improved by up to 14.09% for Adobe* Illustrator and 9.73% for Microsoft* Publisher, which frequently reload hot modules. The translation time drops by 28.85% for Microsoft* Publisher and 28.71% for Adobe* Illustrator, and the memory consumption drops by 59.46% and 24.04% respectively. The overhead brought by the translation reuse engine is almost ignorable, which merely accounts for less than 0.2% of the translation time. As dynamic module loading/unloading is accepted and practiced by more and more desktop application developers, we believe that the module-aware translation is an indispensable feature for dynamic binary translators targeting to real-life desktop applications.

9. REFERENCES

[1] Baraz Leonid, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang and Yigal Zemach, “IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems”, 36th Annual International Symposium on Microarchitecture, 2003.

[2] Eric R. Altman, Kemal Ebcioglu, Michael Gschwind and Sumedh Sathaye, “Advances and Future Challenges in Binary Translation and Optimization”, Proceedings of the IEEE Special Issue on Microprocessor Architecture and Compiler Technology, November 2001.

[3] Eric R. Altman, David Kaeli, and Yaron Sheffer, “Welcome to the opportunities of Binary Translation”, IEEE Computer 33(3), March 2000.

[4] R.L. Sites, A. Chernoff, Kirk, M. Marks, and S. Robinson, "Binary Translation," Comm. ACM 36 (2), Feb. 1993.

[5] Kemal Ebcioglu, Erik R. Altman, Michael Gschwind and Sumedh Sathaye, “Dynamic Binary Translation and Optimization”, IEEE Transactions on Computers 50(6), June 2001.

[6] M. Gschwind and E. Altman, “Optimization and precise exceptions in dynamic compilation”, in the Proceedings of the 2000 Workshop on Binary Translation, October 2000.

[7] Youfeng Wu, Mauricio Breternitz, Justin Quek, Orna Etzion, Jesse Fang, “The accuracy of initial prediction in two-phase dynamic binary translators”, in the Proceedings of the International Symposium on Code Generation and Optimization, 2004.

[8] SPEC CPU2000 http://www.specbench.org/osg/cpu2000

[9] Sysmark 2004 http://www.barpco.com

[10] Dehnert, J.C.; Grant, B.K.; Banning, J.P.; Johnson, R.; Kistler, T.; Klaiber, A. and Mattson, J., “The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges” in the Proceedings of the International Symposium on Code Generation and Optimization, 2003.

[11] Cindy Zheng and Carol Thompson, “PA-RISC to IA-64: Transparent Execution, No Recompilation”, IEEE Computer 33(3), March 2000.

[12] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavall and John Yates “FX!32: A Profile-Directed Binary Translator”. IEEE Micro(18), March/April 1998.

[13] J.M. Anderson, et al, “Continuous Profiling: Where Have All the Cycles Gone?,” ACM Transactions on Computer Systems, Vol. 15, No. 4, November 1997, pp. 257-390.

[14] Kim Hazelwood, Michael D. Smith, “Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems”, in the Proceedings of the International Symposium on Code Generation and Optimization, 2004.

[15] Paul J. Drongowski, David Hunter, Morteza Fayyazi, David Kaeli, “Studying the Performance of the FX!32 Binary Translation System”, in the Proceedings of the 1st Workshop on Binary Translation , Newport Beach, CA, Oct. 1999.

[16] The Java Virtual Machine Specification. http://java.sun.com/docs/books/vmspe

[17] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banjeria, “DYNAMO: A Transparent Dynamic Optimization System”, Programming Language Design and Implementation, June 2000.

[18] Kemal Ebcioglu and Erik R. Altman “DAISY: Dynamic Compilation for 100% Architectural Compatibility”, Proceedings of the 24th Annual Symposium on Computer Architecture, June 1997.