ASCMS: an Accurate Self-Modifying Code Cache Management Strategy in Binary Translation

Anzhan Liu
School of Software Engineering
Zhongyuan University of Technology
Zhengzhou, 450000, China
e-mail: liuanzhan@126.com

Wenqi Wang*
School of Computer Science
Zhongyuan University of Technology
Zhengzhou, 450000, China
e-mail: ww7109@163.com

Abstract—**Self-modifying code poses potential problems in binary translation. When the original source code had written by itself, the translated code block from source code must be retranslated. Self-modifying code must be accurately emulated by the runtime. To improve translation efficiency of self-modifying code, this paper design and realize a new policy named ASCMS for self-modifying code cache management. The ASCMS provides a precise positioning to a translated block, not to a trace or the whole code cache. Through the simulation experiments, The ASCMS has 3.95 times increase to self-modifying code in binary translation.**

Keywords- self-modify code; binary translation; code cache management; ASCMS

I. INTRODUCTION

Self-modifying code has been used to hide the internals of a program for a long time. For example, a floppy disk drive which access instruction 'int 0x24' would be written into the executable's memory image after the program started executing[1]. Many academic literature describes how the self-modifying code to prevent reverse engineering [1, 2, 3].

Self-modifying code is very well suited for those applications, In which reverse engineering is assumed to be one of the main problems, [4].

However, in most of the binary translation research, Self-modifying code is one of the biggest challenges. One of the most affect of self-modifying to binary translation is the code cache management. The most common method of code cache is to flush all the translated blocks when self-modifying occurs.

To accelerate self-modifying code translation, this paper designed code cache policy name ASCMS, and realized fundamental ASCMS prototype system. The test show that with the increasing of the counts that self-modifying code occurs, the ASCMS strategy acceleration ratio is basically stable at around 3.95, when there have 11 Flush Block counts (FB) in Code cache; Under the situation that FB is different, ASCMS strategy acceleration ratio increased with FB added, as to say with FB increasing, the advantage of ASCMS appear more obvious predominant. ASCMS can be used to improve translation efficiency to self-modifying code.

II. BINARY TRANSLATION

Binary translation is used to run binary code on a different ISA. There are several binary translators on the binary translation history.

Some projects have focused solely on translation from fixed ISA, Some not.

There have been many projects about binary translation, including UQBT[5], Embra, BOA[6][7], DAISY[8][9], FX!32 [10][11][12], Transmeta Crusoe[13] and Intel Pentium4 [14]. Because these projects exist a large, but sometimes incomplete, body of knowledge, we believe that our work has a firm basis. UQBT in conjunction with NJMC suggest ways of describing, if not specifying processors such that translation tools can be automatically generated. Results provided by BOA and Crusoe can guide our choice of translation time. Under software and hardware deployment conditions, FX!32 and Pentium 4 both provide some successful examples. Binary translation has been widely discussed in paper like [11] .

Even for the simplest application, binary translation also has a great time overhead. Embra has slowdown to a 4x for the fastest case, from 20x-100x in previous systems. We believe the translation performance more easily accepted by users, if more advanced techniques is applied to binary translation. In addition, we intend to stress the use of run-time optimizations. In general, ISA gap is the main reason for the reduction of translation performance between the host and targets.

III. SELF-MODIFYING CODE

Self-modifying code is one of the biggest challenges in binary translation, although Self-modifying code is not in common.

Self-modifying code technique can dynamically generate code at runtime, so to protect code against both static and dynamic analysis. For example, Madou etal. [15] propose a technique named PRNG(pseudo random number generator) where functions are constructed prior to their first call at runtime to protect the constant "edits" against dynamic analysis.

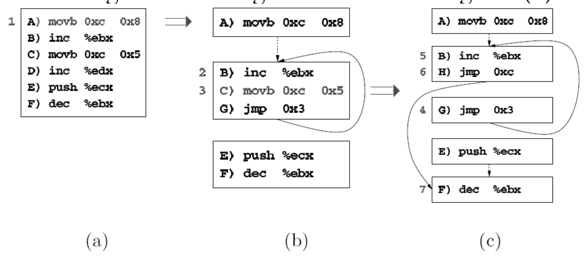
Decryption at runtime technique can also be dynamically generated code. If the decryption key rely on other code, then it is self-modifying code technique.

As a example, here have chosen the example of self-modifying code from paper [16] .the example like this:

Address	Assembly	Binary
0x0	movb 0xc 0x8	c6 0c 08
0x3	inc %ebx	40 01

```
0x5        movb 0xc 0x5        c6 0c 05
0x8        inc %edx            40 03
0xa        push %ecx           ff 02
0xc        dec %ebx            48 01
```

Here gives out a single basic block in Figure 1(a).



```
1 A) movb 0xc  0x8
  B) inc  %ebx
  C) movb 0xc  0x5
  D) inc  %edx
  E) push %ecx
  F) dec  %ebx
```

```
  A) movb 0xc  0x8
2 B) inc  %ebx
3 C) movb 0xc  0x5
  G) jmp  0x3
  E) push %ecx
  F) dec  %ebx
```

```
  A) movb 0xc  0x8
5 B) inc  %ebx
6 H) jmp  0xc
4 G) jmp  0x3
  E) push %ecx
7 F) dec  %ebx
```

Figure 1. (a)construction before execution, (b)after the first write instruction A, (c)after the second write instruction C[16]

However, the instruction A change D into G, resulting in a new control flow graph (b). Next instruction B is executed. Then C changes itself into H as shown in graph(c). Then G back to B and H. Then F is executed. Therefore the trace is A-B-C-G-B-H-F. We can see that "inc %ebx" could not be executed repeatedly.

IV. Code Cache Management Challenges

The code cache is a memory area in which translated blocks are stored. Several replace policies [17] are useed to replace a translated block in code cache, when the code cache is full.

For code caches have the software-managed nature, spent maintaining the cache would impact the performance of the executing application directly. In this section, we will look at the unique code cache management challenges. And we present software-managed code caches in the later sections must overcome.

A. *Variable-Sized Translated Blocks.*

Hardware caches usually handle fixed length blocks. On the other hand, the translated block which created by dynamic binary translation systems often vary significantly in size, from a few bytes to several thousand bytes [18]. This is an important property that distinguishes code caches from hardware caches. This wide range of translated block sizes results in significant fragmentation in the code cache under many replacement algorithms (e.g., LRU). If translation systems compact or remove these fragmentation, it will lost a lot of performance at run time, and the chaining of translated block makes the complexity of the problem. Therefore, some replacement strategy prefer a code cache-management technique which avoids fragmentation or simplifies the process of compaction, such as FIFO replacement [19].

B. *Translated Block Chaining.*

Translated block chaining is one of the major reasons that degrade performance in a dynamic binary optimizer [20][21]. In one code cache region, off-trace branches are patched to jump directly to their target regions. However, eviction of a translated block can change the code cache region into an inconsistent state, unless the link pointers used to implement chaining are updated to reflect the eviction. To solute this problem, to provide a side table for back pointers is a generic method. When a translated block in code cache will be removed, the back-pointer table will be looked by the eviction mechanism to determine another translated blocks to be linked to the eviction candidate. Unfortunately, back-pointer table carries run-time overhead for lookups and takes up memory that is used for code cache. If using flushing the entire code cache as management strategy, then the back-pointer table will be unnecessary. Policy of flushing all of the translated blocks in the entire code cache name full flush policy. Along with the translated blocks all link information will be flushed too. Except full flush all other policies for code cache-management will have additional performance loss, for supporting link removal will require time and memory.

C. *Translated Block Regeneration Overhead.*

There is completely different for Servicing software code cache and hardware cache misses, because in anyther alse elements that have been translated and stored in software code cache have not another copy. The method of regenerating the previously cached translated code block is used to service conflict and capacity misses in software code cache. For example, copying of the code block into the code cache, and updating of hash tables, and updating of translated block links information. Significant run-time overhead is the result. this process takes on the order of in the DynamoRIO system [22] where 5 thousand instructions for a typical SPEC 2000 translated block are tested. In order to reduce this high code update overhead, Dynamic optimization systems designers will spare no effort for maximizing code cache hit rate.

V. Design and Implementation of ASCMS

A. *Discover Self-modifying Code*

ASCMS can discover the self-modifying code by detecting signal. The process of discovering self-modifying code describes as follow. First, the system through the initial loading module loads source binary file to the allocated memory page. Taking into account the memory protection mechanism using a page as a basic unit, in the Linux OS, a page size is 4096 bytes, ASCMS system in the period of allocating memory space needs to be done a certain special treatment, which is made up of memory opened the first address is 4096 as a whole several times. After sources binary code load into the memory, the system memory protection mechanisms will open up the pages of memory protection, which is only allowed to read. When self-modifying code occurs, the Linux operation system will have a signal SIGSEGV. Through catch the signal, ASCMS can discover self-modifying code. And the signal SIGSEGV must to be disposed, otherwise, operation system will report a segment fault. Segment fault will lead to system termination.

B. *Deal with Self-modifying Code*

ASCMS deal with self-modifying code starts with the capturing the signal SIGSEGV. To avoid system interrupted by the signal, ASCMS changes the process of the signal's response. In the process, the Linux core function reset SIGSEGV signal procession The new process of the signal modifies the protection mode allows the code to modify itself. Then the source code was modified into a new version. ASCMS has added the source code back-up mechanism to precious locating the modified area code.

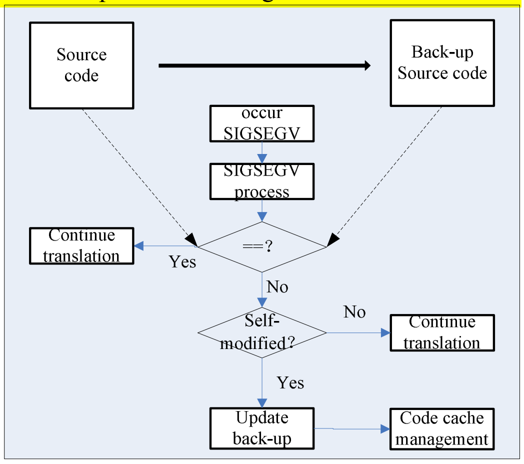


Figure 2. the Process of Dealing with SMC

As shown in Figure 2, a backup mechanism is adopted in the management of translator for source code. When the source code in binary is modified, the system will trigger signal SIGSEGV. Through comparing source code and back-up source code, the system can find whether the code is modified by itself. If self-modifying has occurred, the system will update the back-up source code and go to the code cache management.

C. *Code Cache Policy*

1) Translation Block

Translation Block (TB) is the basic block in the translation process, for short TB. Translator implements the translated code by side with translating it. The basic unit of code that the translator disposes is TB. The TB is an instruction sequences that end of control transfer (such as a branch, or call Jump command). ASCMS system describes every translation block with a structure. For each block, the description includes the information of pc which point out the TB's location in source code. And the description of the block also includes information which trace the TB belong to via trace ID. ASCMS system described TB using the data structure that is fully compatible with QEMU TB description, is added to the Trace ID information. This help the system to find a basic block and corresponding trace in the code cache with a high efficiency. For all the TB in translation process, a TB set is formed.

2) Trace

Logically speaking, Code cache is a memory space include some code segments which are organized into Trace chain. From the physical sense, code cache is only a contiguous memory space for storing the translated block. For each of these translated blocks, it has a first address and in the rear has links or return, in order to link to the next block or return back.

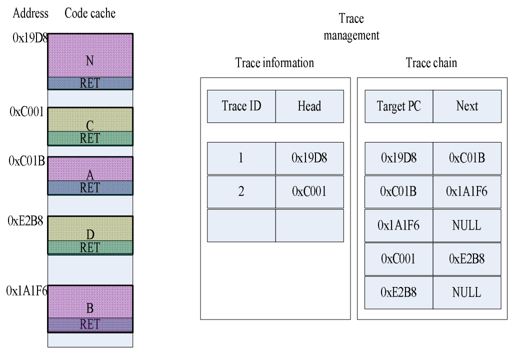


Figure 3. Trace Information and Trace Chain

Figure 3 is described the management of code cache with only two trace chains. The left is the Code cache in the actual physical distribution of the code, a total of five translated block, every block has a hex physical address, such as block A with the start address 0x1A1F6. Each block end have a return instruction. After a block is executed over, the translator will find the next block according to the information in Trace management. If a translated block is found, it will be executed, otherwise, the translator will continue to translate a new block. In the table of trace information, there is information of TraceID and Head. Every TraceID has a corresponding Head which point the first block address in Trace chain. Once the translator finds the Head of a trace, then it can find all the next blocks easily according to the information in table of trace chain. when the translator is to run a trace, Trace manager will start from the head of a trace to start.

Figure 3 displayed the two Trace chains, N- A -B -NULL and C - D -NULL, the corresponding ID of the first chain is 1, and the second is 2. Trace information table gives the information of ID and Head, Head pointing to the corresponding Target PC in the table of trace chain. In the Trace information In the table of trace information the TraceID 1 corresponding Head is 0x19D8. In the table of trace chain, the corresponding next information of 0x19D8 is 0xC01B. In the code cache block N has the first address 0x19D8, and the block A has the first address 0xC01B. This means that the block N link to A. In accordance with the above method, Figure 3 displayed the two Trace: N-A-B-NULL and C-D-NULL. Trace information stored through an array, with the subscript of the array as Trace ID, the value of the array elements is the head value of a trace. To array Trace [], as shown in Figure 3 Trace [1] = 0x19D8, Trace [2] = 0xC001. Through this method can easily find the actually block's Target PC from TraceID.

3) Block Replace

For the self-modifying code cause change of the binary code which is loaded in the memory, the relevant translated block must be retranslated again. The retranslated measure is divided into two kinds of methods. One is replace the dirty block at the primary area. The other is opening up a new memory space to place a new translated block. If the size of the new translated block is not more than the old, the new translated block are placed into the primary area. Such as described in (b) of figure 4. If the size of the new translated block is larger than the old, the new translated block are placed into a new area. Such as described in (c) of figure 4. Otherwise the new block is placed into a new space.

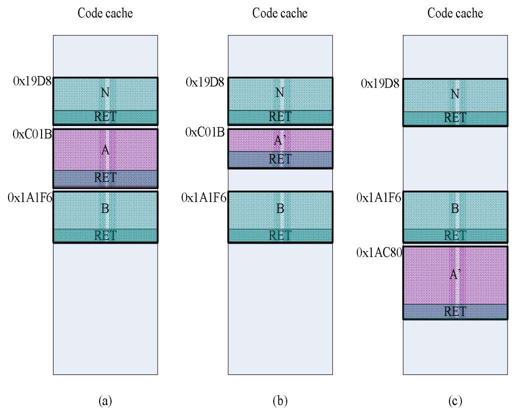


Figure 4. Translation Block Replace Strategies

Figure 4 describes the replacement of two different strategies, when self-modifying code arose the block A in (a) must be retranslated, a new translated block A' is created by the translator. In (b), the size of A' is less than A, so the A' is placed into the area where A is there. Thus the A' has the same first address as A and no more configure is to be set to the trance management. If the trace N-A-B-NULL has formed, trace management is necessary to amend the relevant information. The advantage of this replacement strategy is that it will not affect the relevant trace information after replacing the dirty block. So does not need to maintenance the trace information. the disadvantage is that it will cause fragment in code cache. In (c), A' is larger than the size of A, so the A' is placed a new space allocated by the system. This cause the first address of block A changed into 0x1AC800. If the trace N-A-B-NULL has formed, it must to be changed into N-A'-B-NULL. And the system has to maintenance the trace information. This is the disadvantage of this strategy.

4) TB Location

Before translating, translator loads the source binary code into memory. To simulate this process ASCMS first opened up a physical page in memory, and then loads a binary code into the page. After this has completed, the system will protect the page, so that when modify occurs the system would trigger a signal SIGSEGV to notice system. At this time the system will compare the the source code with the back-up source code. According to the result of comparison, the system can calculate the offset of the modified address in a page.

To locate a TB, ASCMS defines a page block mapping table. Every page includes several TB. Through the mapping table, ASCMS locates a TB, after has calculated the offset of the TB.

Pageoffset	Size	&TB
0	187	137166848
323	363	137167171
886	108	137167734
. . .	. . .	. . .

Figure 5. Page Block Mapping Table

As shown in Figure 5, when the system detect self-modifying happen, the system starts the comparison mechanism and calculates the modified offset of the page. Assumed to be page offset 582, the system searches the page block mapping table with dichotomy. Because of $323 \leq 582 < 323 + 363$, the corresponding entry in the table can be found, thus ASCMS can get the pointer of the TB. In figure 5, the corresponding address is 137167171.

VI. Testing and Analysis

ASCMS proposed a treatment of code cache when the self-modifying code occurs during the process of translation. In dealing with self-modifying code, ASCMS adopts memory signal and back up loaded source binary source code mechanism. When self-modifying takes place, ASCMS locates the precious position of the modified translated block, the system only retranslates the block that has been modified. Through this mechanism, the system needs not to flush all the translated blocks in code cache. Compared to flush strategy, ASCMS increase a certain storage expenses, such as memory for backing up source code and mapping table. Testing of ASCMS is mainly concentrated in the time efficiency. Here through contrast method to FLUSH strategy to evaluate advantage and disadvantage of the ASCMS strategy. In order to facilitate the description, the definition of strategy accelerate ratio is gived out.

Definition 1: In a time of self-modifying, the used time ratio of FLUSH and P strategy is defined to strategy accelerate ratio of P strategy. Using A(P)to express.

This can be described with formula (1)

Formula (1):

A (P) = the used time of FLUSH strategy / the used time of P strategy.

By definition 1 it can see that strategy A (FLUSH) is the value of 1, because the A (FLUSH) = the used time of FLUSH strategy / the used time of FLUSH strategy = 1.

As the using time of FLUSH strategy depends on the magnitude of translated blocks cleared in code cache, the strategy accelerating ratio of P is relying on the block's quantities in code cache.

If FB is expressed the magnitude of translated blocks cleared by FLUSH strategy in code cache, E(P,FB) is expressed the used time of P strategy, A(P) can be expressed to A(P,FB). The formula (1) will be expressed to formula (2).

Formula (2):

$A(P, FB) = E(FLUSH, FB)/E(P, FB)$

Definition 2: In N times of successive self-modifying, the average of P strategy accelerate ratio is defined to N times average accelerate ratio of P strategy. A (P, FB, N) is described with formula (3)

Formula (3):

A (P, FB, N) = （A1+A2+…+AN）/N.

Formula (3) can also to be expressed to formula (4).

Formula (4):

A （P， FB， N） = （A1 （P， FB） +A2 （P， FB） +…+ AN（P， FB））/N

Next, the accelerate ratio of ASCMS strategy is tested and analysised. Through several tests of FLUSH and ASCMS strategy's time efficiency at a time of self-modifying, to analysis the relative performance of the ASCMS strategy.

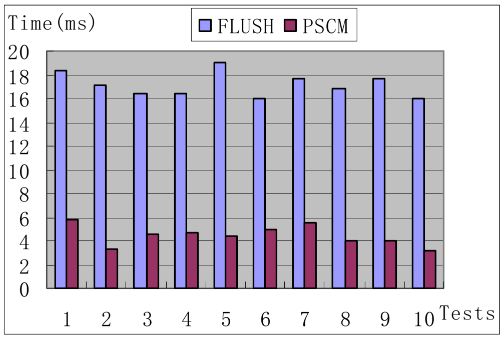


Figure 6. FLUSH and ASCMS Performance Contrast of once Self-modifying

According to the analysis of chapter five, FLUSH strategy removed an average of 10.76 blocks in code cache. For ASCMS simulation system can not be set the number of block to decimal, the time implement efficiency as the FB is 11. Figure 6 is comparison chart of E (FLUSH, 11) and E (ASCMS, 11), it is 10 times repeat test results when once self-modifying code takes place. From figure 6, we can see that ASCMS strategy has a obviously advantage in time efficiency than FLUSH strategy. On average, the A(ASCMS,11) is about 3.86, which explains ASCMS strategy has 3.86 times advantage than FLUSH on the efficiency of time.

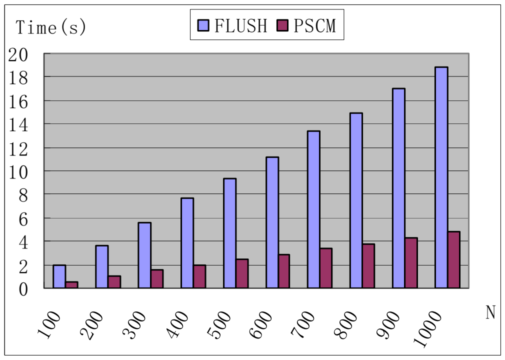


Figure 7. FLUSH and ASCMS Performance Contrast of N times Self-modifying

Figure 7 is given out the E (FLUSH, 11, N) and E (ASCMS, 11, N) comparison chart. From the chart, the time increases with the N, whenever which strategy is adopted FLUSH or ASCMS. For A(ASCMS,11,N) is the ratio of E(FLUSH,11,N) and E(ASCMS,11,N), Figure 8 will describe the relationship between A and N.

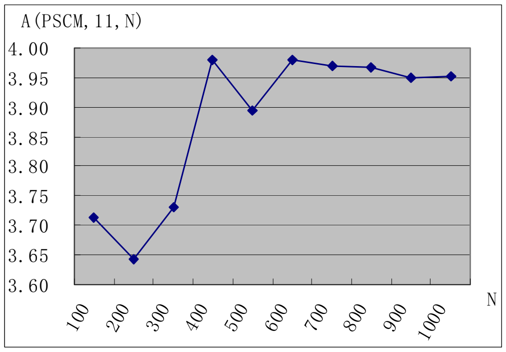


Figure 8. the Relationship between A and N

Figure 8 shows that, when FB is 11, the strategy accelerate ratio of ASCMS is between 3.6 and 4.0. With the increase of N, the A(ASCMS, 11, N)is basically stable at around 3.95. This explains that the strategy accelerate ratio of ASCMS strategy is about 3.95, when FB is 11.

The above results are tested as the FB is 11. the efficiency of FLUSH strategy depends on the counts of translated block in code cache. in order to comprehensively analyze the performance of ASCMS strategy, we must test the time efficiency in different value of FB. Not all the values of FB could be tested, So we test the value from 5 to 17. Figure 9 is the Relationship between time efficiency and FB for FLUSH and ASCMS strategy.

From the figure 9 we can see that with the increase of FB, the times of FLUSH strategy gradually increases, In contrast, the efficiency times of FB in the ASCMS strategy is essentially the same. This also shows ASCMS does not increase consumption of time with counts of block increased in code cache.

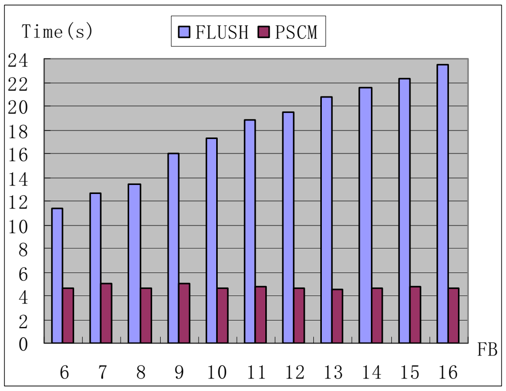


Figure 9. the Relationship between Time and FB

Figure 10 show that, with the increase of FB, the A is also increased. With the continued increase in FB, the strategy accelerate ratio will also continue to increase. This shows that the ASCMS strategy will have more advantage with the increasing counts of block in code cache.

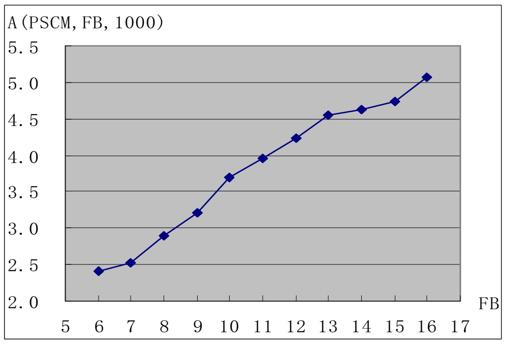


Figure 10. the Relationship between A and FB with the same N

VII. CONCLUSIONS

The testing to ASCMS is indicated: With the increasing of the counts that self-modifying code occurs, the ASCMS strategy acceleration ratio is basically stable at around 3.95, when the quantity FB of translated blocks in Code cache is 11; Under the situation that FB is different, ASCMS strategy acceleration ratio increased with FB added, as to say with FB increasing, the advantage of ASCMS appear more obvious predominant. ASCMS can be used to improve translation efficiency to self-modifying code.

Our future work is to extend our research and to apply the ASCMS into a real binary translator such as QEMU. We will continue to investigate various strategies of code cache management in a translator for self-modifying code. Finally, these code cache management schemes will be directly implemented in a dynamic binary translator, we will get more each policy's specific details about performance details.

REFERENCES

[1] 1 D. Aucsmith. Tamper resistant software: an implementation. Information Hiding, LNCS, 1174:317–333, 1996.

[2] 1 Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting selfmodication mechanism for program protection. In Proc. of the 27th Annual International Computer Software and Applications Conference, pages 170–181, 2003.

[3] 1 M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. Information Security Applications, LNCS, 3786:194–206, 2005.

[4] 1 C. Cifuentes and K. Gough. Decompilation of binary programs. Software – Practice & Experience, 25(7):811–829, 1995.

[5] 1 Cristina Cifuentes, Brian Lewis, and David Ung. Walkabout a retargetable dynamic binary translation framework.

[6] 1 Michael Gschwind et al.,"Dynamic and Transparent Binary Translation,"Computer,Vol 33,No 3, March 2000,IEEE Computer Society Press,pp 54-59.

[7] 1 Michael Gschwind,Erik Altman,"Inherently Lower Complexity Architectures using Dynamic Optimization," Proc.Workshop on Complexity Effective Design in conjunction with ISCA-2002,Anchorage,AK,May 2002.

[8] 1 K.Ebcioglu and E.Altman,"DAISY:Dynamic Compilaton for 100 Percent Architectural Compatibility," Proc.ISCA24,ACM Press,New York,1997,pp.26-37.

[9] 1 Michael Gschwind, Kemal Ebcioglu, Erik Altman, and Sumedh Sathaye. Daisy/390: Full system binary translation of ibm system/390,1999.

[10] 1 Paul J. Drongowski, David Hunter, Morteza Fayyazi, and David Kaeli. Studying the performance of the fx!32 binary translation system.

[11] 1 R.J.Hookway and M.A.Herdeg,"Digital FX!32:Combining Emulation and Binary Translation," Digital Technical J.,Vol.9,No.1,1997,pp.3-12.

[12] 1 R Hookway,"DIGITAL FX!32 running 32-Bit x86 applications on Alpha NT", Proceedings IEEE COMPCON 97.Digest of Papers.San Jose,CA,USA.IEEE Comput.Soc.23-26 Feb.1997.

[13] 1 J. C. Dehnert, B. K. Grant, J. P. Banning,R. Johnson, T. Kistler, A. Klaiber,and J. Mattson. The transmeta code morphing software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In International Symposium on Code Generation and Optimization. CGO 2003. San Francisco, CA, 2003.

[14] 1 Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Doug Carmean, Alan Kyker, and Patrice Roussel. The microarchitecture of the pentium 4 processor. Intel Technology Journal,2001.

[15] 1 M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software protection through dynamic code mutation. In J. Song,T. Kwon, and M. Yung, editors, The 6th International Workshop on Information Security Applications (WISA 2005), volume LNCS 3786, pages 194–206. Springer-Verlag, August 2006.

[16] Bertrand Anckaert, Matias Madou, and Koen De Bosschere. A Model for Self-Modifying Code ,2006

[17] K. Hazelwood and M. D. Smith. Code cache management schemes for dynamic optimizers. In 6th Workshop on Interaction between Compilers and Computer Architectures, pages 102–110, February 2002.

[18] HAZELWOOD, K. AND SMITH, J. E. Exploring code cache eviction granularities in dynamic optimization systems. In 2nd International Symposium on Code Generation and Optimization. Palo Alto, CA, 89–99. 2004.

[19] HAZELWOOD, K. AND SMITH, M. D. Code cache management schemes for dynamic optimizers.In 6th Workshop on Interaction between Compilers and Computer Architectures. 102–110. 2002.

[20] BALA, V., DUESTERWALD, E., AND BANERJIA, S. Dynamo: A transparent dynamic optimization system. In Conference on Programming Language Design and Implementation. 1–12. 2000.

[21] CMELIK, B. AND KEPPEL, D. Shade: A fast instruction-set simulator for execution profiling. ACM SIGMETRICS Performance Evaluation Review 22, 1 (May), 128–137. 1994.

[22] HAZELWOOD, K. AND SMITH,M. D. 2003. Generational cache management of code traces in dynamic optimization systems. In 36th International Symposium on Microarchitecture. San Diego, CA. 169–179.