Efficient and Effective Misaligned Data Access Handling in a Dynamic Binary Translation System

JIANJUN LI, Institute of Computing Technology
Graduate University of Chinese Academy of Sciences
CHENGGANG WU, Institute of Computing Technology
WEI-CHUNG HSU, National Chiao Tung University

Binary Translation (BT) has been commonly used to migrate application software across Instruction Set Architectures (ISAs). Some architectures, such as X86, allow Misaligned Data Accesses (MDAs), while most modern architectures require natural data alignments. In a binary translation system, where the source ISA allows MDA and the target ISA does not, memory operations must be carefully translated. Naive translation may cause frequent misaligned data access traps to occur at runtime on the target machine and severely slow down the migrated application.

This article evaluates different approaches in handling MDA in a binary translation system including how to identify MDA candidates and how to translate such memory instructions. This article also proposes some new mechanisms to more effectively deal with MDAs. Extensive measurements based on SPEC CPU2000 and CPU2006 benchmarks show that the proposed approaches are more effective than existing methods and getting close to the performance upper bound of MDA handling.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Code generation; optimization; run-time environments*

General Terms: Management, Performance

Additional Key Words and Phrases: Optimization, misaligned memory access, binary translation

ACM Reference Format:
Li, J., Wu, C., and Hsu, W.-C. 2011. Efficient and effective misaligned data access handling in a dynamic binary translation system. ACM Trans. Architec. Code Optim. 8, 2, Article 7 (July 2011), 29 pages.
DOI = 10.1145/1970386.1970388 http://doi.acm.org/10.1145/1970386.1970388

1. INTRODUCTION

Binary Translation (BT) [Sites et al. 1993] is a technique commonly used to migrate application binaries from one ISA to another [Chernoff et al. 1998; Baraz et al. 2003; Bellard 2005; Ebcioğlu and Altman 1997; Scott et al. 2003; Transitive Technologies

---

This article is an extension of the article entitled "An Evaluation of Misaligned Data Access Handling Mechanisms in Dynamic Binary Translation Systems" which appeared in *the Proceedings of the International Symposium on Code Generation and Optimization (CGO'09)* in 2009.
This article is supported in part by the National Science and Technology Major Project of China (2009ZX01036-001-002), the National Natural Science Foundation of China (NSFC) grant 60736012, the National Science Council grant 99-2218-E-009-008-MY2, the Innovation Research Group of NSFC grant 60921002, the Chinese National Basic Research grant 2011CB302504 and the National High Technology Research and Development Program of China (No. 2007AA01Z110).
Authors' addresses: J. Li and C. Wu, Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China; email: {lijianjun, wucg}@ict.ac.cn; W.-C. Hsu, Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan; email: hsu@cs.nctu.edu.tw.
Permission to make digital or hard copies of part or all 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 show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2011 ACM 1544-3566/2011/07-ART7 $10.00
DOI 10.1145/1970386.1970388 http://doi.acm.org/10.1145/1970386.1970388

Ltd. 2001]. When the target ISA is the same as the source ISA, BT can also be applied to implement dynamic optimization [Bala et al. 2000; ke Chen et al. 2000; Bruening et al. 2003; Duesterwald 2005], dynamic speculation and parallelization [Packirisamy et al. 2006], dynamic instrumentation [Nethercote and Seward 2007; Bungale and Luk 2007], and dynamic software security enforcement [Qin et al. 2006]. Some architectures do not have alignment restrictions where data operands must start at addresses that are multiples of the data's natural size [Hennessy and Patterson 2006]. For example, X86 allows Misaligned Data Accesses (MDA). The implementation of such architectures may provide hardware support to generate multiple memory operations in completing the misaligned memory access. Alternatively, the implementation may choose to generate a misaligned access trap and let software handle the required memory accesses. For architectures that disallow MDA, such as MIPS, ALPHA, IA-64, and most modern RISCs, misaligned data accesses using regular memory operations will always generate misaligned access traps. Since the cost of misalignment exception is very high, frequent MDAs will severely slow down the execution of the application. Misaligned traps usually would not occur because the compilers for modern RISCs ensure data are properly allocated to fulfill the alignment requirements. The compiler-managed data allocation would preclude MDA from happening at runtime. However, in binary translation systems, if the source architecture allows MDA so that the data in the original binary were not properly aligned, the directly translated memory operations may generate an MDA trap. Therefore, in a binary translation system, when the source architecture allows MDA, memory operations must be translated carefully to minimize the performance impact from frequent misalignment traps.

Existing binary translation systems have been dealing with MDA using different mechanisms. For example, the Transmeta code morphing system relies on hardware support [Dehnert et al. 2003; Coon et al. 2006]. QEMU [Bellard 2005] directly translates the source memory operations into a sequence of safe target memory operations. The FX!32 [Chernoff et al. 1998; Hookway and Herdeg 1997] uses static profiling to identify source memory operations that are likely to incur MDA and translates only such candidates into a code sequence free of misalignment traps. The IA-32 EL system [Baraz et al. 2003] exploits dynamic profiling to collect candidate memory operations for optimization during the hot code optimization phase. The direct translation method used in QEMU is simple but incurs significant overhead for every source memory operation that references more than one byte. A profiling-based approach is an effective way to selectively translate problematic MDA memory instructions so that other memory operations do not need to pay such overhead. Static profiling is simpler than dynamic profiling, but is less adaptive. If the MDA behavior changes from a profiled run to the actual runs, MDAs not identified by static profiling will pay a high cost. Dynamic profiling is more adaptive. However, the current implementation of dynamic profiling is often based on interpretation and instrumentation which prevent it from a full-blown implementation. Existing systems often take a compromised approach and implement "one shot" dynamic profiling, where dynamic profiling is done once instead of being continuous. This would leave MDA instructions which are not detected from the first shot of dynamic profiling unhandled and later cause significant performance drops from frequent misalignment traps.

This article evaluates different methods for MDA handling in existing binary translation systems and proposes a few new mechanisms to more effectively translate memory instructions. It makes the following contributions:

—a comprehensive study on efficient handling of MDA in BT systems for the X86 architecture;

—a thorough analysis of existing mechanisms which handle MDA, including a direct method, a static profiling method, and a dynamic profiling method;
—a few new mechanisms, such as an exception handler-based method, a combination of dynamic profiling, and exception handler-based method, and a continuous profiling method, are proposed. These new mechanisms can adaptively handle MDA according to behavior changes of the program. For some applications, the performance gain can be greater than 10% when compared with existing mechanisms.

The rest of this article is organized as follows: Section 2 gives a brief overview of MDA in a BT system and Section 3 introduces basic MDA translation strategies. Section 4 discusses how to identify MDA candidates and their associated attributes including static profiling, dynamic profiling, and continuous profiling. It also proposes a more adaptive mechanism to handle MDA based on the use of a misalignment exception handler. Section 5 describes some additional optimizations for MDA code generation. Section 6 provides the experimental framework in which the misalignment handling mechanisms are evaluated and discusses performance results. Section 7 discusses related work. Section 8 summarizes and concludes this work.

2. MDA IN A BT SYSTEM

How often do MDAs occur in a BT system if the source ISA has no alignment restrictions? A basic understanding of the behavior of misaligned memory operations could help us in designing more effective handling methods. We have collected data from a binary translation system where the source ISA is X86 and the target ISA is Alpha. Table I lists the number of MDAs encountered by the SPEC CPU2000 and SPEC CPU2006 benchmarks (with ref input set) which were compiled for X86 by the pathscale2.4 compiler on an X86/Linux system. In the table, NMI stands for the number of misaligned instructions. Those instructions have incurred misaligned data accesses at runtime. NMI is a static measurement. Ratio is the number of dynamic MDA instances divided by the total number of memory accesses. Table I shows that some programs have very frequent MDAs (e.g., 188.ammp with 43.12% of all memory operations are MDAs, and 179.art, 38.33%) and some programs have essentially no MDAs (e.g., 462.libquantum and 473.astar). On average, a program may incur 9.5 billion MDAs. If each MDA is handled by the misaligned access trap handler, which may cost nearly 1K cycles [Baraz et al. 2003; Drongowski et al. 1999], then the average overhead could be as high as 9.5K seconds on a 1 GHz machine. However, as Table I shows, the majority of the benchmarks have very low ratios of MDAs, so any MDA handling method that significantly burdens normal memory operations should not be considered. Furthermore, the NMI of those programs with a high ratio of MDAs is from a few hundred to a couple thousand. Comparing the NMI to the total number of memory instructions in SPEC benchmarks, MDA instructions are still a small portion which means the binary translator should not indiscriminately treat all memory operations as MDA candidates.

Conventional wisdom has it that even in computers that allow misaligned data access, programs compiled with data aligned would still run faster [Hennessy and Patterson 2006]. Therefore, it seems that Independent Software Vendors (ISV) may prefer to release their X86 binaries with compiler optimization to enforce aligned memory accesses. It is true that many compilers for the X86 architecture do support optimizations that enforce data alignments, such as the Intel CC, the Pathscale, and the GCC compiler. However, this optimization is usually not turned on by default. In addition, such alignment optimizations were not used for released SPEC benchmark numbers on the X86 architecture. We have tested the performance impact of data alignment on X86 machines using the benchmarks which have a high frequency of MDAs (ratio>0.1%). Figure 1 shows the performance with alignment optimization (using the Pathscale and

Table I. MDAs in SPEC CPU2000 and CPU2006

Benchmarks	NMI	Num of MDAs	Ratio	Benchmarks	NMI	Num of MDAs	Ratio
164.gzip	80	406,431,686	0.52%	400.perlbench	77	1,469,188,415	0.26%
175.vpr	134	2,762,730	0.01%	401.bzip2	45	82,641,256	0.01%
176.gcc	154	37,894,632	0.06%	403.gcc	53	32,624	0.00%
181.mcf	16	1,649,912	0.02%	429.mcf	10	883,518	0.00%
186.crafty	20	4,950	0.00%	445.gobmk	76	1,741,956	0.00%
197.parser	16	291,054	0.00%	456.hmmer	127	13,757,509	0.00%
252.eon	3096	8,523,707,162	9.63%	458.sjeng	9	1,303	0.00%
253.perlbmk	270	148,689,820	0.23%	462.libquantum	9	435	0.00%
254.gap	14	1,128,048	0.00%	464.h264ref	96	138,883,221	0.01%
255.vortex	90	12,361,950	0.03%	471.omnetpp	394	6,303,605,195	3.37%
256.bzip2	44	25,233,188	0.04%	473.astar	32	758	0.00%
300.twolf	98	441,176,894	0.92%	483.xalancbmk	53	5,749,815,279	1.60%
168.wupwise	132	9,682	0.00%	410.bwaves	602	99,916,961,773	12.67%
171.swim	284	49,605,944	0.03%	416.gamess	424	13,073,700	0.00%
172.mgrid	78	1,772,430	0.00%	433.milc	3825	67,272,361,837	12.09%
173.applu	306	2,243,041,896	1.60%	434.zeusmp	3484	87,873,451,026	4.14%
177.mesa	54	9,370	0.00%	435.gromacs	197	123,577,765	0.01%
178.galgel	5282	492,949,052	0.27%	436.cactusADM	48	1,745,161	0.00%
179.art	1024	21,244,446,764	38.33%	437.leslie3d	205	23,645,192,624	2.54%
183.equake	30	524	0.00%	444.namd	103	10,516,106	0.00%
187.facerec	112	6,240,872	0.01%	450.soplex	538	13,446,836,143	5.71%
188.ammp	1134	73,194,953,020	43.12%	453.povray	918	36,294,822,277	8.30%
189.lucas	64	17,383,280	0.02%	454.calculix	139	478,592,675	0.02%
191.fma3d	398	5,383,029,436	3.36%	459.GemsFDTD	3304	31,740,862	0.00%
200.sixtrack	1324	8,673,947,498	4.21%	465.tonto	1748	38,717,125,228	3.80%
301.apsi	356	1,568,299,486	0.86%	470.lbm	8	7,124,766,678	1.14%
481.wrf	92	49,694,156	0.00%	482.sphinx3	115	3,118,790,131	0.31%
Average	597	9,525,126,313	1.44%				

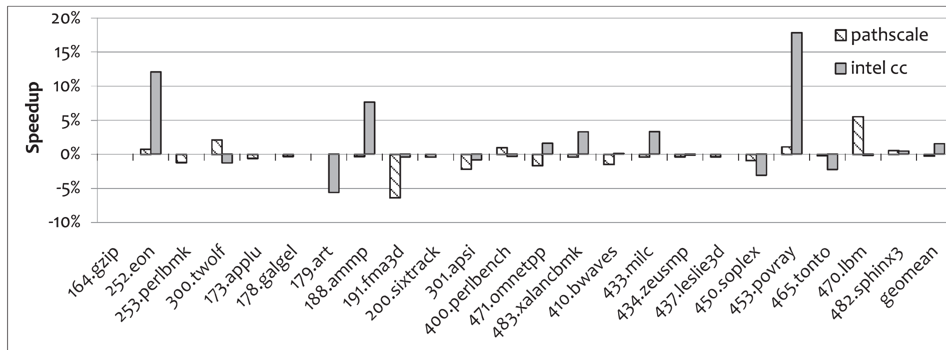

Fig. 1. Performance with alignment optimization flags for Pathscale and Intel CC.

the Intel CC compilers) on X86 machines. There were no significant performance advantages with data alignment (1% for Pathscale and 1.8% for Intel CC, on average). The performance gains from aligned data accesses could be outweighed by the increased data working set size. This may explain why many released X86 binaries were not compiled with data aligned.

We have observed frequent MDAs in the X86 shared libraries, such as libc.so.6 and libgfortran.so.6. We have noticed that more than 90% of MDAs that occurred in 164.gzip, 400.perlbench, and 483.xalancbmk (100% for 164.gzip, 99% for 400.perlbench and 91% for 483.xalancbmk) are actually from shared libraries. Even if some ISVs release their binaries with data alignment enforced, as long as the application uses

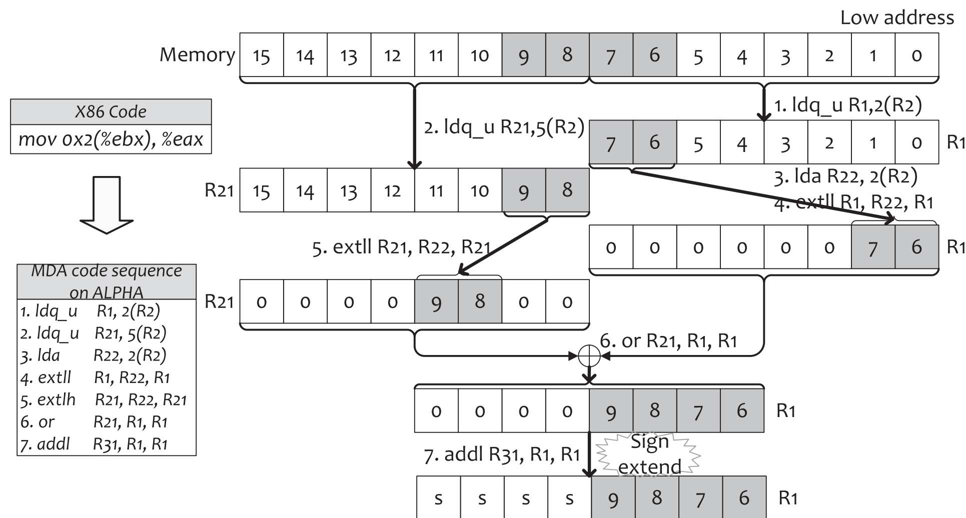

Fig. 2. MDA code sequence on Alpha.

the shared libraries, frequent MDAs may still occur at runtime. Therefore, it is critical that a BT system handles MDA efficiently if the source ISA allows misaligned data accesses.

3. TRANSLATION OF MEMORY INSTRUCTIONS

For any data access that is greater than a single byte, misaligned data accesses could potentially occur and would cause a misaligned access trap at runtime on machines with alignment requirements. Some architectures with no hardware support to handle MDAs usually provide a code sequence (we call it MDA code sequence in this article) to access misalignment data without triggering misalignment exceptions. In fact, the same MDA code sequence can be used by the misalignment exception handler to access the misaligned data. On the Alpha architecture, there are some special instructions to support the handling of misaligned data accesses, such as the ldq_u (Load Quadword Unaligned), extll (Extract Longword Low), and extlh (Extract Longword High) instructions. On MIPS and Itanium architectures, there are also several special instructions to support MDAs such as ldl/ldr on MIPS and shrp on Itanium. Figure 2 gives an example of translating an X86 instruction to the MDA code sequence on Alpha[1] [Compaq 2002; Intel 2009].

In this example, register %eax and %ebx in X86 are mapped to register R1 and R2 in the Alpha binary respectively and register 21-30 of Alpha are used as temporal registers in BT. We assume the address stored in R2 is 4-byte aligned. The right side of Figure 2 illustrates how the MDA code sequence works. The first two ldq_u instructions are used to retrieve the data into two temporal registers, then the designated data are merged into the destination register, R1, and an addl instruction is used to sign-extend the longword to quadword. Note that in Alpha, R31 means value zero.

Both Alpha and MIPS have provided MDA code sequences with special instructions such as ldq_u and extll. For architectures with no such ISA support, a simple sequence

[1]In the example, both X86 and Alpha are running in little-endian mode. If the endianness of the host and guest architecture are different, the MDA sequence should be changed. However, the endianness would not affect the handling of MDAs; it only affects how to translate the memory reference instructions.

of load byte instructions combined with some logic operations could serve the same purpose but at a higher cost [Smith and Nair 2005].

While the MDA code sequence is much longer than a simple load word instruction, it avoids the possible misalignment traps. Therefore, if a memory operation is likely to be an MDA, it should be translated into the MDA code sequence. If it is unlikely to be an MDA, it might be better to allow it to remain as a regular memory operation and leave the rare misaligned access to exception handling [Adams and Agesen 2006]. This is the basic idea of MDA handling. The challenging issues here are: (a) how to determine the probability of the memory operation being an MDA, (b) determining if the behavior of the MDA operation is stable, and (c) if the behavior tends to change, to determine if it is changing in a predictable pattern. In the following section, we give several alternatives in translating nonbyte memory instructions into MDA code sequence.

3.1. Direct Method

As Table I shows, failing to translate nonbyte memory instructions into MDA code sequence could possibly lead to numerous misaligned traps. A simple approach is to indiscriminately translate all nonbyte memory instructions into MDA code sequence. We call this the direct method. The direct method can be expensive since a single memory operation after translation would take several instructions (and many more cycles) to execute. Since some applications have almost no MDA, direct translation will significantly slow down such programs.

3.2. Selective Translation

One variation to direct translation is to first identify those memory instructions that would never cause misaligned accesses and bypass generating MDA code sequence for them. Techniques to discover and predict such candidates can often be guided by misalignment profiles. Different profiling approaches will be discussed in the next sections.

3.3. Multiversion Code Generation

For many nonbyte memory instructions, even if they cause misaligned data access, the frequency could be relatively low. For example, one memory instruction may be executed one billion times but incur MDA only one million times. For such a memory instruction, translating it into MDA sequence would incur overhead 999 million times. On the other hand, not translating them into MDA sequences would pay for one million expensive trap handlings. To solve this problem, the binary translator could generate two versions of native code: one version of the MDA code sequence and the other version remaining as a single memory operation. At runtime, the version to execute would be selected according to the actual memory address referenced. Intuitively, multiversion code would work better except for the cases where the memory instructions were extremely biased toward misaligned accesses. In fact, the extreme cases occur quite often in SPEC benchmarks, as shown in Table I.

3.4. Adaptive Code Generation for Phased Behavior

While many memory instructions are biased towards aligned or misaligned accesses, they may be biased towards one type over a significant period of time then switch to another type. For example, one instruction may have all aligned memory accesses during the first half of the run and then start to turn into MDAs. For this phased behavior, even multiversion code is undesirable because for the first half, the multiversion code must pay for the extra address checks and the increased code size.

In a typical binary translation system, once the source architecture instructions are translated into native binary, the translated code stays in the code cache and remains

unchanged. However, in order to adapt to the aforementioned phased behavior, a BT system may need to adopt an adaptive code generation policy.

A full-fledged adaptive code generation requires continuous monitoring of the access patterns and switch to different code generation dynamically. The cost of instrumentation to monitor the access patterns can be prohibitive. However, a simplified adaptive code generation can be cost-effectively implemented. For example, a BT system could initially assume memory instructions are referencing aligned addresses. Once MDA is detected for this instruction, the translator can then change it to MDA code sequence or multiversion code. In this simplified approach, it catches only one type of phase change: initially aligned and then misaligned. This simple approach has been shown very effective.

4. IDENTIFY MDA CANDIDATES

The best translation of a memory instruction depends on its behavior listed as follows.

—If it never has any misaligned operand addresses, the preferred translation would be a respective native load/store instruction.
—If it has frequent misaligned operand addresses, the preferred translation would be the MDA code sequence.
—If it swings between aligned and misaligned operand addresses, the preferred translation would be two-version code.
—If it exhibits phased behavior, for example, it has aligned operand addresses for a long period of time, and then switches to misaligned addresses, the preferred translation should be adaptive code.

The binary translator needs information about the behavior of each memory instruction to determine the preferred translation. Usually, such information may be collected either by interpretation or binary instrumentation, and the information feedback could be static (collect profile using a training input and feedback to the static translation), dynamic (collect profile as the program runs and feedback to the dynamic translator), or continuous (using accumulated profile collected from previous runs to feedback to the next translation). We have also included the exception handler in the process of collecting misaligned profile.

4.1. Static Profiling-Based Method

Some BT systems rely on static profiling to identify MDA candidates. A profiling run with training input set collects information on MDA (e.g., addresses of MDA instructions). Guided by this profile, the binary translator generates the MDA code sequence for the memory instructions with high MDA probability [Chernoff et al. 1998]. Figure 3 depicts the mechanism of the static profiling-based method.

This mechanism is effective if the training input is representative and can reflect the real execution behavior of the program. However, this is not always true. Furthermore, many programs allocate data dynamically, and it is difficult to predict the likelihood of alignment for dynamically allocated data references. Therefore, with static profiling, MDAs may still occur frequently. If a MDA is not identified by static profiling but actually happens in a hot loop, the performance penalty could be very high.

4.2. Dynamic Profiling-Based Method

In order to overcome the limitation of static profiling, many dynamic binary translation systems adopt a two-phase translation approach. The first phase is for profile collection. In this phase, the binary is either interpreted or translated with instrumentation to collect profile information. The second phase is the actual translation phase. In this phase, hot regions identified from the first phase are retranslated and optimized.

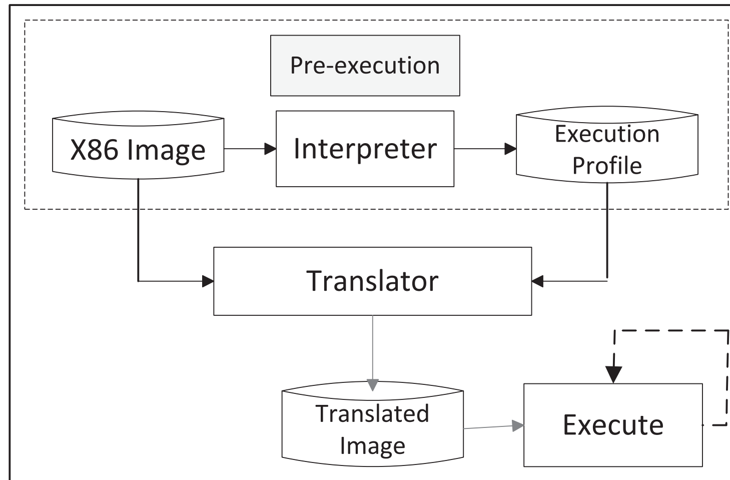

Fig. 3. Static profiling-based method.

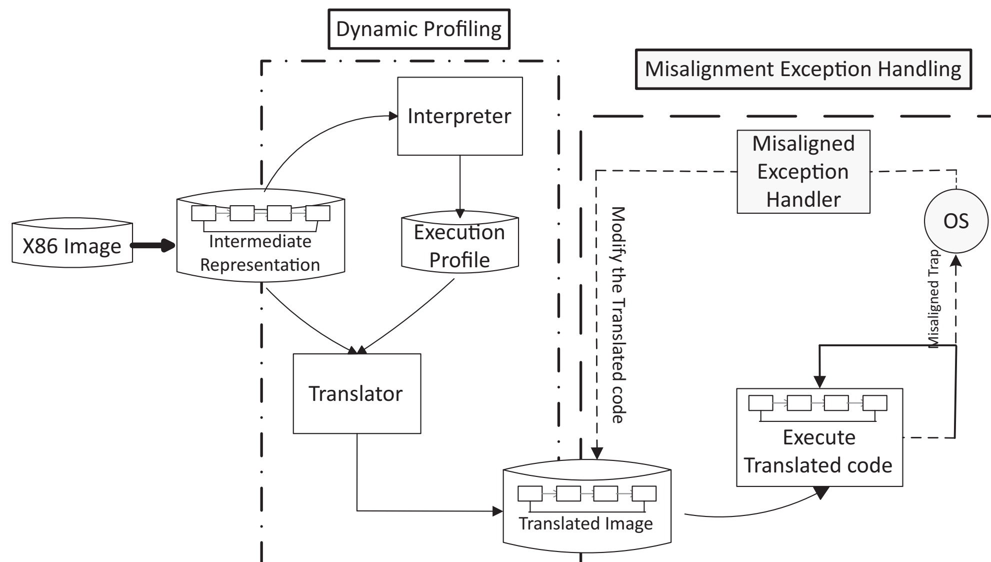

Fig. 4. Dynamic profiling and exception handling mechanisms in MDA handling.

Consider IA-32 EL, for example, is a two-phase dynamic binary translator. Its first phase, called cold code translation phase, is designed to be simple with minimal optimizations and uses instrumentation to collect execution profiles and identify hotspots. Its second phase, called hot code translation phase, retranslates and further optimizes those hotspots [Baraz et al. 2003].

The dynamic profiling method is based on the two-phase binary translation approach. As depicted in the left side of Figure 4, all memory reference instructions are interpreted or translated with light instrumentation in the first phase. When an MDA occurs at runtime, the detailed misalignment information is recorded, including the address of the MDA instruction, the type of misalignment (e.g., the current address is aligned to what type of boundaries: byte, halfword, word, or double word), and the

access pattern of this MDA instruction (e.g., always misaligned, alternate patterns, or mixed patterns). Then during the hot code translation phase (i.e., the 2nd phase), the recorded information guides the code selector to determine whether MDA code sequences should be generated. In the experiments which we show in Section 6, we generate MDA code sequence for a memory access instruction if the instruction has performed MDA once during the profiling stage. In contrast to static profiling, dynamic profiling responds better to behavior changes with different inputs, and it does not require a separate profiling process. However, if the behavior change occurs after the translation of MDA code, the performance would suffer (either from MDA traps or from unnecessary MDA code sequence). Our experiments show that increasing the hot code threshold can effectively reduce the MDAs by more accurately capturing those MDA candidates. However, to eliminate a majority of MDAs in some applications, the threshold must be set so high that profiling overhead becomes excessive. For example, the threshold for 410.bwaves must be as high as 266K if we want to eliminate most MDAs. This threshold setting is application dependent, and we could imagine some applications would require an extended profiling phase. Therefore, this brute-force approach is not acceptable.

4.3. Exception Handling-Based Method

A major drawback of the dynamic profiling method is that its profiling phase must be relatively short because the longer the profiling phase, the greater the overhead. Therefore, if the MDA of an instruction does not show up during the profiling phase, it may not have a chance to get translated into MDA code sequence. To remedy this weakness, we have introduced an exception handler-based method as illustrated in the right side of Figure 4. In the initial translation, we assume all memory references are naturally aligned and translate them into normal memory instructions. Once a misalignment exception is raised in the translated code, the OS calls the misalignment exception handler registered by the binary translation system. In the misalignment exception handler, the following steps will be taken.

- —Obtain and analyze the instruction that incurs misalignment exception with the context information of the exception point.
- —Generate the MDA code sequence for that offending memory instruction.
- —Allocate memory (usually called code cache by BT systems) to store the code sequence.
- —Patch the offending memory operation to a branch instruction jump to the MDA code sequence stored in code cache, and insert a branch instruction back into the block at the end of the MDA code sequence.

To minimize the impact of MDA, the instruction that incurs MDA was patched the first time an MDA exception occurred in the translated code.

The left side of Figure 5 shows the translated native code on Alpha when a 4-byte load incurs misalignment exception at runtime [Compaq 2002]. The code modified by the misalignment exception handler is given in the right side of Figure 5. In the figure, pc1 represents the address of the offending memory operation, and pc2 is the start address of the MDA code sequence. After the exception handler deals with it, the original instruction in pc1 is replaced with a branch instruction which jumps to pc2, and at the end of the MDA code sequence, a branch instruction is inserted to direct the execution back to the next instruction of pc1.

With this method, instructions with MDA showing up late in execution can be caught. In fact, the exception handler-based method implements a simplified adaptive code generation; it assumes all memory instructions were accessing aligned addresses to start with. When a misaligned trap is taken, the translator assumes this instruction will have more MDA for the remainder of the execution, so it translates this instruction into

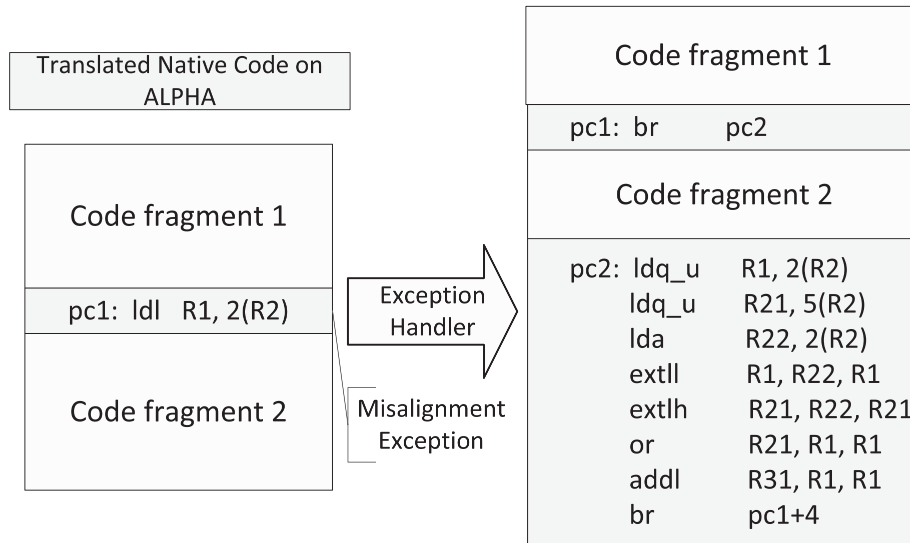

Fig. 5. Example code sequence of the exception handling-based method.

MDA code sequence. Ideally, a more adaptive code generation would insert instrumentation code to continuously monitor whether the behavior of the memory instruction has changed over time. However, the cost of this approach is too high to be profitable. A full-fledged adaptive code generation may require hardware support.

The exception handler-based method works better when it combines with either static profiling or dynamic profiling. Intuitively, this exception handler-based approach only pays one trap handling cost for each MDA candidate. As Table I indicates, most MDA-intensive SPEC programs have several hundred to a few thousand MDA instructions. Intuitively, the overhead for handling this many exceptions is acceptable. However, each trap handling will involve calling the binary translator and patching the code. This is a relatively expensive context switch that a dynamic translation system attempts to avoid. It could incur cache cold misses (the need to reload instructions from the translator and native code, for example). If this method is combined with profiling, then many of the MDA candidates can be handled together in one shot and leave those hard-to-catch, late-coming MDA to the exception handler. This combination drastically reduces the overhead of switching back and forth between the native execution and the binary translation. It also allows the binary translator to better organize the translated code, further improving the code locality.

4.4. Continuous Profiling Method

Static profiling can be more effective if profiles from multiple input sets can be merged or accumulated. For example, in FX!32, the binary translation benefit from continuous profiling is used to discover new hot spots not covered by previous runs. Our exception handler-based method can collect information on new MDA candidates not covered by previous profiling. This information could provide feedback to subsequent runs. It avoids the need to have a separate profiling process. This mechanism of continuous profiling is illustrated in Figure 6.

This approach uses our new exception handler to conduct continuous profiling essentially for free. It significantly reduces profiling overhead and can benefit from continuous accumulating profiles from previous runs. This can have a greater coverage on MDA instructions. One downside of this approach is it tends to generate MDA code

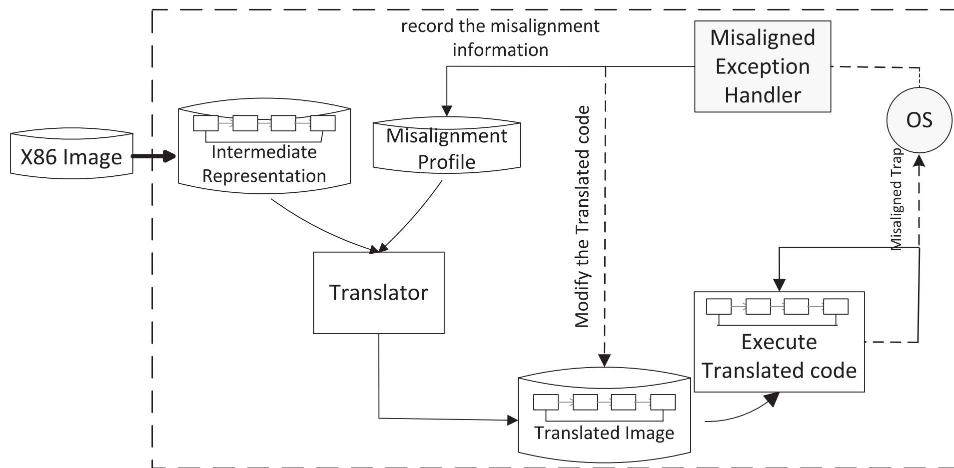

Fig. 6. Continuous profiling handling mechanism.

sequence too early. As discussed in Section 4, some MDA instructions may initially have all aligned accesses for a while and then start to incur misaligned accesses. Generating MDA code sequence too early may induce unnecessary overhead. To avoid this drawback, when we profile the MDA instructions, we also record one additional attribute which indicates whether this MDA is considered early arrival or late arrival. Early or late is determined by how much code has been generated in the code cache. With this early/late attribute, the translator will only translate those MDA candidates as early arrivals. For late arrivals, the translator will leave them untouched and let the exception handler catch the first MDA trap if there is one.

5. ADDITIONAL TRANSLATION OPTIMIZATIONS

Other than the basic MDA translation mechanisms (Section 3) and the MDA identification methods (Section 4), some variations on related optimizations are discussed in this section.

5.1. Granularity for Multiversion Code

What granularity should the binary translator use to generate multiversion code? A simple approach is to generate two-version code for each memory instruction. This approach checks the operand address for alignment and executes the MDA code sequence only when it is actually needed. Consider a basic block with multiple memory instructions. This code generation requires one check for each instruction. After analyzing the applications, we observed that most of MDAs occurred in hot loops and the addresses of MDAs usually followed the same pattern. In this case we can generate multiversion code based on the basic block granularity. For example, a loop is translated into two versions: one version with all regular loads and another one always using MDA code sequence. The memory reference address of the first MDA instruction is checked to determine which version to execute. At basic block granularity, fewer checks are executed. However, this works effectively for programs with regular behavior. For programs with irregular behavior, generating multiversion code for each instruction may work better.

5.2. Full-Fledged Adaptation

So far, we focus more on how to turn a memory operation into the MDA code sequence. What if after we generate the MDA code sequence, the original memory operation no longer has MDA? Should we convert the MDA code sequence back to normal memory

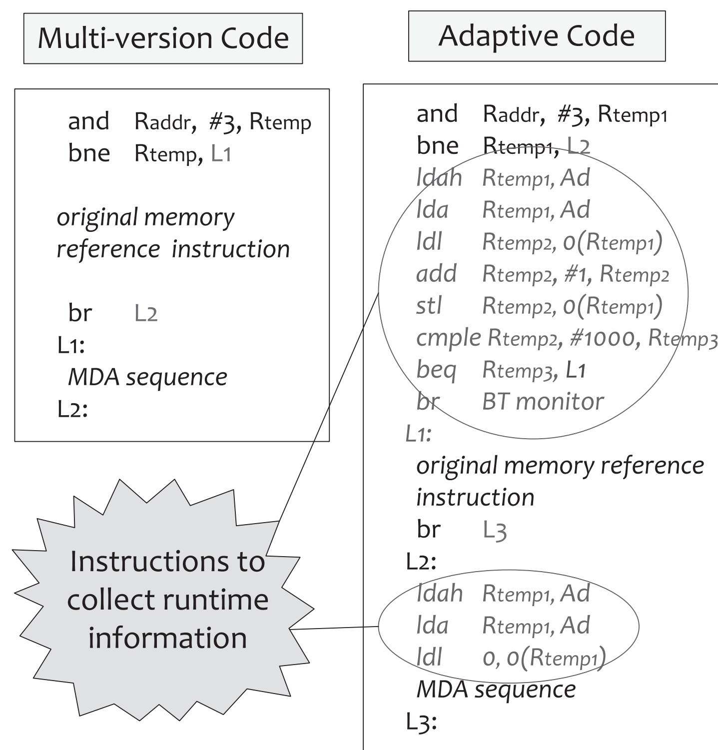

Fig. 7. An example of adaptive code method on Alpha architecture.

operation? This method may be more effective than multiversion code method, but its implementation cost could be prohibitive. Figure 7 compares the multiversion code method and this truly adaptive method. As we can see in Figure 7, the adaptive method needs about ten instructions (including 3 memory access instructions and 2 branch instructions) to collect runtime information which is used to determine if we should return the MDA code sequence back to original memory operation. Furthermore, even if we return the MDA code sequence back to original instructions, only two instructions (one logic instruction and one branch instruction) are saved. Taking all factors we discussed before into account, we believe this seemingly more adaptive method may not be worth pursuing.

5.3. Code Rearrangement

One drawback of the exception handling-based method is that the code locality is decreased after patching each MDA instruction to a jump. This may lead to increased instruction cache misses and result in significant performance loss. To overcome this shortcoming, a method to rearrange the generated code is adopted [Hazelwood and Smith 2006]. An example of this method is illustrated in the right-hand side of Figure 8. When working with profiling methods (static, dynamic, or continuous), a majority of MDA instructions are handled at once so the performance impact of code rearrangement is reduced.

5.4. Retranslation

Dynamic profiling can identify many frequent MDA instructions in a block so that they can be turned into MDA code sequences at the first translation instead of being handled

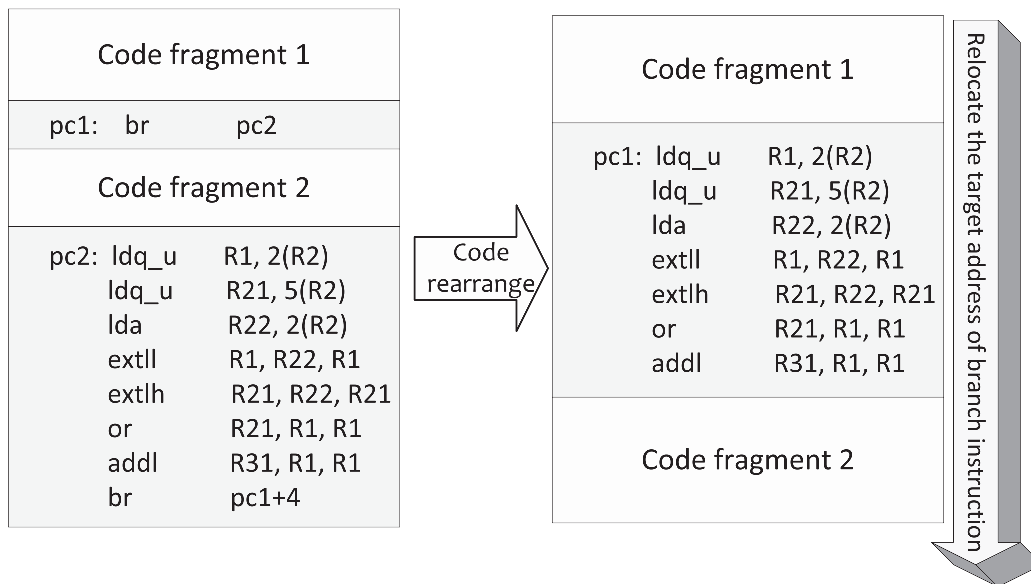

Fig. 8. Example of code rearrangement method.

one by one through the exception handler. However, for programs that have frequent behavior changes, the initial translation based on dynamic profiling may likely be ineffective. When such cases are identified, our BT system can simply invalidate the translated code for a basic block [Hazelwood and Smith 2006] and restart the dynamic profiling and translation process for this block. In the exception handler, we record the number of MDA exceptions that occurred in each block. When the number of MDA exceptions in a block reaches a threshold, the original translated native code of the block is invalidated and the process of retranslation is initiated. This is somewhat similar to the code cache flush policy employed in Dynamo except that Dynamo [Bala et al. 2000] flushes the entire code cache while our BT invalidates translated code at block granularity.

This retranslation approach is another simplified adaptive code generation but at a basic block granularity.

6. PERFORMANCE EVALUATION

In this section we first describe the machine and benchmark sets used in performance evaluation of different misalignment handling methods. Then we conduct an estimation of the overhead for each misalignment handling method based on the cost of MDA code sequence and exception handling. In the estimation model, we only consider the cost of instruction executed. We use the actual occurrence of misaligned accesses and memory operations as parameters to calculate the estimated cost. We also estimate the upper bound of performance improvement which is the case that the system pays the cost of MDA code sequence only if a misaligned access occurs. Since this estimation is not accurate in that the impact of caches, the execution pipeline, and the interaction between the runtime management system and native execution are not included, we also conduct actual runs on the Alpha machine to verify the performance results.

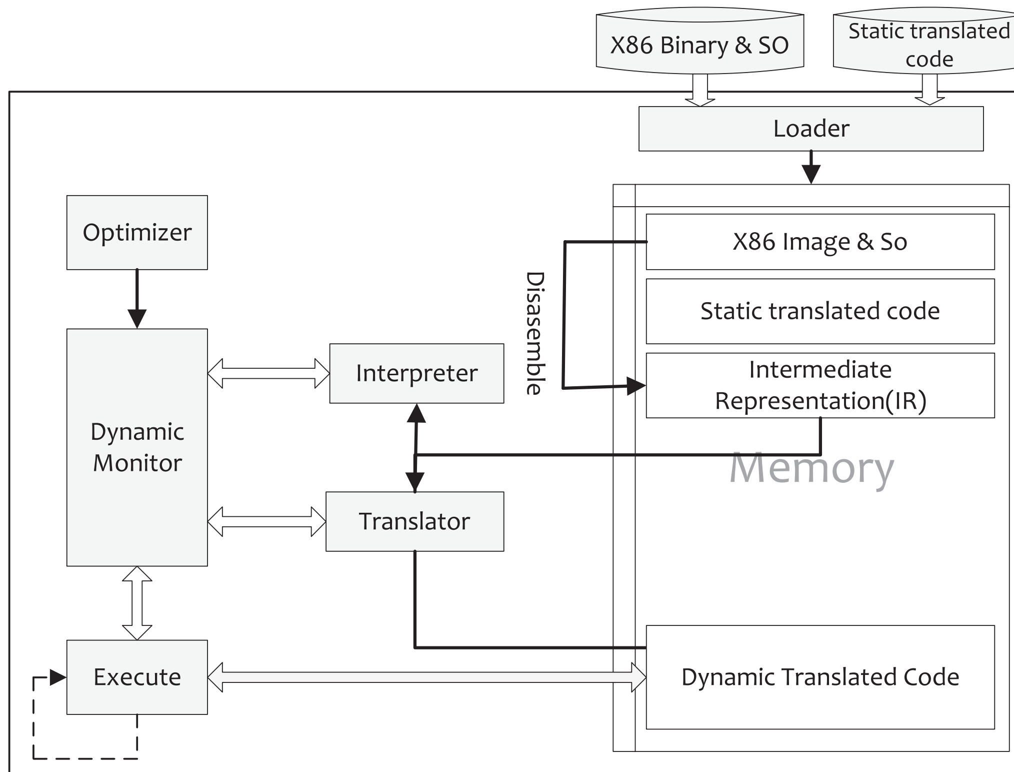

Fig. 9. Framework of DigitalBridge.

6.1. Target Machine

Various misalignment handling methods are evaluated on a 1-way Alpha ES40 machine running CentOS 4.2. The processor's on-chip cache hierarchy consists of separate 64KB, 2-way set associative L1 instruction and data caches and a unified 2MB direct-mapped L2 cache. The memory size of the machine is 4GB.

6.2. Experiment Framework

We have adopted the BT framework of DigitalBridge to evaluate all different MDA handling mechanisms. DigitalBridge, which is developed by us, is a dynamic translator which migrates X86 binaries to Alpha architecture. Similar to FX!32 [Chernoff et al. 1998], DigitalBridge supports X86 binary codes to run on Alpha machines. Unlike FX!32, DigitalBridge uses online instead of offline translation. DigitalBridge also employs the two-phase translation approach. In the first phase, it executes the source binary code on a basic block granularity and collects profile data to guide optimizations in the second phase. DigitalBridge is a fully functioning system. Figure 9 shows the major components of DigitalBridge. In this article, all examples of MDA handling will assume X86/Alpha as the guest/host machine model.

In the figure, *X86 Binary & SO* is the source binary code and the shared libraries which the source program uses; *Static translated code* is the translated code of source program which is generated by the static binary translator; *IR* is the intermediate representation of the source binary code, it is generated by the dissembler in our system; *Dynamic translated code* is the native binary code generated by the translator

at runtime; *Interpreter* interprets the source binary code and collects profile data to guide optimization in the translation phase; *Translator* translates the source binary code and stores the translated code into code cache; *Optimizer* optimizes the binary code generated by the translator; *Dynamic monitor*, which is the runtime monitor of the binary translation system, manages the components of the BT system, and handles the exceptions, signals.

6.3. Benchmarks

The full suite of SPEC CPU2000 and SPEC CPU2006 benchmarks were used for evaluation in this article. However, the impact of MDA handling in BT systems is determined by the frequency of MDAs in applications. Therefore, we report performance results only for the benchmarks that have a significant number of MDAs (the ratio of MDA is greater than 0.1%) according to Table I. In our final set, 23 benchmarks are included, of which 11 are from SPEC CPU2000 (164.gzip, 252.eon, 253.perlbmk, 300.twolf, 173.applu, 178.galgel, 179.art, 188.ammp, 191.fma3d, 200.sixtrack, 301.apsi) and 12 are from SPEC CPU2006 (400.perlbench, 471.omnetpp, 483.xalancbmk, 410.bwaves, 433.milc, 434.zeusmp, 437.leslie3d, 450.soplex, 453.povray, 465.tonto, 470.lbm and 482.sphinx3). The performance is represented as normalized ratios in the figures. We have also reported the geometric mean of the ratios for the 23 benchmark programs.

All benchmarks are compiled using Pathscale 2.4 on an Intel Xeon machine, and the compiler options are -Ofast -m32 -LNO:simd=0.

6.4. Misalignment Handling Methods

We have evaluated the following MDA handling methods.

Original. This is the method of doing nothing by leaving all MDA at runtime to exception handling.

Direct Method. This is to translate all nonbyte memory instructions to MDA code sequence. It is used by the QEMU dynamic binary translator [Bellard 2005].

Static Profiling. This approach takes profiles from the training input set. The dynamic translator takes this profile to select MDA candidates.

Dynamic Profiling. This approach used dynamic profiles collected during the profiling phase to guide the selection of MDA candidates. In the profiling phase, we set the threshold to 50.

Exception Handling. This approach assumes all memory instructions were aligned. If a memory instruction incurs an MDA exception, the handler will generate MDA code sequence for it.

DPEH+MV. This approach combined exception handling method with dynamic profiling. It uses dynamic profiling to handle all MDA candidates during the translation phase. In the profiling phase, the misaligned access frequency of each MDA instruction is collected. Unless an MDA instruction is extremely biased, it will be translated into multiversion code. In addition, all late-coming MDA instructions will be translated into multiversion code by the request from the exception handler.

Continuous Profiling+MV. This approach uses continuous profiling to accumulate MDA profiles from multiple runs and feedback to the next dynamic translation. It also combines with the exception handling method in that whenever one memory instruction incurs an MDA exception, this instruction will be translated into multiversion code. Similar to DPEH+MV method, if an MDA instruction has more aligned accesses according to the accumulated profile, multiversion code will be generated.

Best. This is the best case that may be achieved in practice. We use the best translation strategy for each MDA instruction according to its real behavior observed from the actual runs with the *ref* input. For example, if the memory accesses of one MDA

instruction are more than 80% misaligned, this MDA instruction is translated into the MDA code sequence; otherwise, it is translated into two-version code.

Ideal. This is an ideal case that the program pays the cost of MDA code sequence only when MDA occurs. This gives the upper bound of performance that we can ever achieve. We use this bound to indicate how close we are to the best we can do and how much room is left for further improvement.

6.5. Performance Estimation Using a Simple Model

We estimate the overhead of MDA handling with a simple model which only considers the number of instructions executed. This is because the performance impact of caches, the stall cycles in pipelines, and the cost of branch mispredictions are much more difficult to model. We will leave such performance factors to be measured in real runs.

In our simple model, we assume there are X aligned accesses and Y misaligned accesses for each memory instruction. X1 is the number of aligned accesses after the 1st MDA. Hence the overhead for MDA handling of each method can be formulated as follows.

$$\text{original} = \sum_{i=1}^{\text{NM}I} Y_i * \mathit{TrapCost} \quad (1)$$

$$\text{direct method} = \text{Number of Non-byte memory operations * MDACost} \quad (2)$$

$$\text{dynamic(static) profiling} = \sum_{i=1}^{NMI_1} (X_i + Y_i) * \text{MDACost} + \sum_{i=1}^{NMI_2} Y_i * \mathit{TrapCost} \quad (3)$$

$$\text{exception handling} = \sum_{i=0}^{NMI} (X1_i + Y_i) * \text{MDACost} + \sum_{i=0}^{NMI} \text{TrapHandlerCost}_i + \text{RetranslationCost} \quad (4)$$

$$\text{DPEH(Continuous)} = \sum_{i=1}^{NMI_1} (X_i + Y_i) * \text{MDACost} + \sum_{i=1}^{NMI_2} (X1_i + Y_i) * \text{MDACost} + \sum_{i=1}^{NMI_2} \text{TrapHandlerCost}_i + \text{RetranslationCost} \quad (5)$$

$$\text{DPEH(Continuous)} + \text{MV} = \sum_{i=1}^{NMI_{11}} (X_i + Y_i) * \text{MDACost} + \sum_{i=1}^{NMI_{12}} (X_i * \text{MVCost} + Y_i * (\text{MDACost} + \text{MVCost})) + \sum_{i=0}^{NMI_2} (X1_i * \mathit{MVCost} + Y_i * (\text{MDACost} + \text{MVCost})) + \sum_{i=1}^{NMI_2} \text{TrapHandlerCost}_i + \text{RetranslationCost} \quad (6)$$

$$\text{best} = \sum_{i=0}^{NMI} ((4 * X1_i > Y_i)?(X1_i + Y_i) * \text{MDACost} : (X1_i + Y_i) * (\text{MDACost} + \text{MVCost})) \quad (7)$$

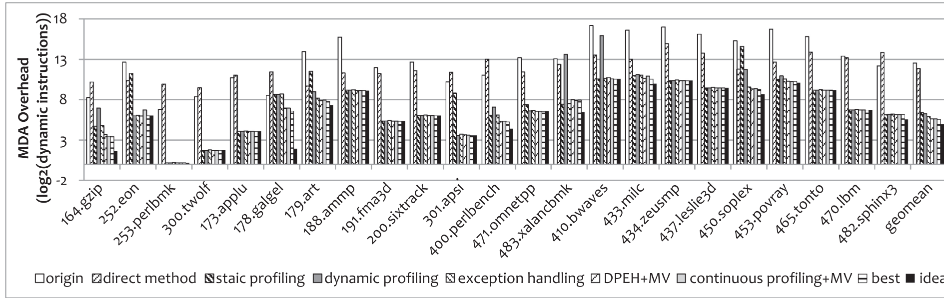

Fig. 10. Estimated MDA overhead for different methods.

$$\text{ideal} = \sum_{i=0}^{NMI} Y_i * \text{MDA cost} \tag{8}$$

In the preceding formulas, NMI stands for the number of MDA instructions; NMI_1 stands for the number of MDA instructions that can be detected by dynamic profiling method (or static profiling method); NMI_2 stands for the number of MDA instructions that cannot be detected by dynamic profiling method (or static profiling method). In formula (6), NMI1 is divided into NMI_{11} and NMI_{12}. NMI_{11} stands for the number of MDA instructions which are translated into MDA code; NMI_{12} stands for the number of MDA instructions which are translated into multiversion code.[2] *Continuous* is short for Continuous profiling method; *MV* is the abbreviation of MultiVersion method; *MVCost* is the cost of multiversion code; *MDACost* is the overhead incurred by MDA code sequence; *TrapCost* is the cost of one misalignment exception while it is handled by OS; *TrapHandlerCost* is the overhead of the misalignment trap handler in our runtime system; *RetranslationCost* is the cost of retranslation process and we count it for different applications and MDA handling methods in our BT system. Because the dynamic profiling overhead is negligible, we do not take it into account in formulas (3), (5), and (6).

In our system, the MDACost is about 10 instructions, the MVcost is about 3 instructions, the TrapCost is about 1K instructions, and the TrapHandlerCost is about 3K instructions. After we put in these parameters, the estimated cost of each benchmark as well as their upper bound is shown in Figure 10.

On average, our proposed approaches (Exception Handling, DPEH+MV and Continuous Profiling+MV) outperform existing ones, and have a performance very close to the upper bound (i.e., the best case) for most of the programs.

6.6. Performance Evaluation on the Real Alpha Machine

This section evaluates different MDA handling methods in actual runs, and the data we present here is an average time of three identical runs.

6.6.1. Original Code. Figure 11 shows the performance loss without MDA handling. The slowdown is normalized to the exception handler-based method. Without MDA handling, the programs will run extremely slow, hence we are only able to obtain the performance data of SPEC CPU benchmarks with the *train* input set. Using the *test* input set, the runtime could be shorter, but using the *train* input set would allow the program to behave more like the reference input set. As shown in the figure, without MDA handling method, some benchmarks suffer significant MDA overhead (3096X for

[2]NMI, NMI_1, NMI_2, NMI_{11}, and NMI_{12} are all static measurements.

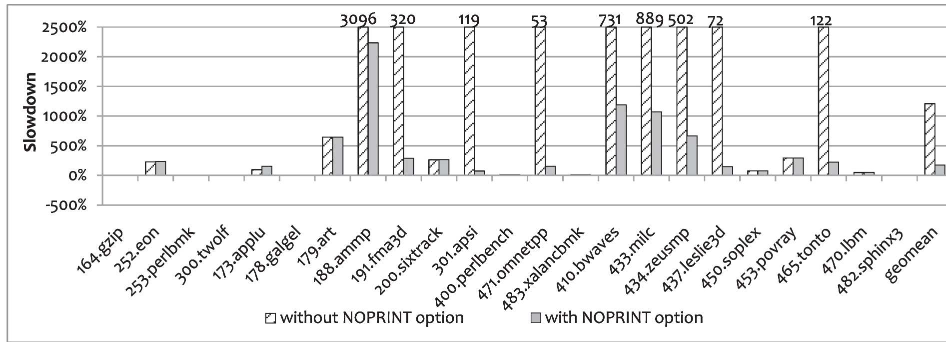

Fig. 11. Performance loss without MDA handling (using train input set) *(baseline is the runtime of exception handling method).*

Table II. The Number of Misaligned and Aligned Accesses of the MDA Instructions (train input set)

Benchmarks	Misaligned	Aligned	Benchmarks	Misaligned	Aligned
164.gzip	27938792	3.48E+08	471.omnetpp	2.89E+09	97174184
252.eon	4.75E+08	20232	483.xalancbmk	7.81E+08	2.09E+08
253.perlbmk	7977553	1420	410.bwaves	6.99E+09	89911986
300.twolf	6512382	445	433.milc	2.37E+09	2.41E+09
173.applu	45425463	3891	434.zeusmp	4.91E+09	3824
191.fma3d	2.77E+09	307309	437.leslie3d	3E+09	3825
301.apsi	29926808	8735	450.soplex	2.14E+08	1.98E+08
178.galgel	43366894	4.49E+09	453.povray	1.16E+09	4.19E+08
179.art	1.11E+09	6.4E+08	465.tonto	1.48E+10	29178451
188.ammp	9.65E+09	5.54E+08	470.lbm	5.04E+08	32
200.sixtrack	2.15E+09	3238284	482.sphinx3	30246134	14654661
400.perlbench	88099763	2.65E+08			

188.ammp, 731X for 410.bwaves, 889X for 433.milc, and 502X for 434.zeusmp). On average the performance degradation is about 12.08X.

We have said that the cost of an MDA is about 1000 cycles in Section 2, so the slowdown should be no more than 1000 times. However, the data shows that 188.ammp is slowed down by 3096X. Our investigation reveals that the program runs relatively slower while the number of MDAs is over one billion. This is because the MDA exception handler of the OS writes a message to the system's log file, and too many messages will cause additional I/O issues. Luckily, there is a syscall in Linux which can disable message writing from the MDA handler. Figure 11 also shows the slowdown with such an option (i.e., NOPRINT). As the data shows, even with the NOPRINT option there is still a significant slowdown for some benchmarks (22.3X for 188.ammp, 11.9X for 410.bwaves, and 10.7X for 433.milc).

Table II shows the number of misaligned and aligned accesses of the MDA instructions (with *train* input set). As the table shows, the programs which have significant performance degradation are exactly the ones which have a large number of MDAs.

6.6.2. Direct Method. This method translates all memory instructions to MDA code sequence. However, Table I shows many programs have almost no misaligned memory accesses for the entire execution. For simplicity, translating all memory instructions into two-version code, one is MDA code sequence and the other is a single memory operation, may significantly reduce the overhead of this direct method. Figure 12 shows the performance gain/loss while generating two versions of code instead of MDA code sequence only. As the figure shows, most benchmarks benefit significantly from two-version code here. The average gain is about 18% with some benchmarks gaining

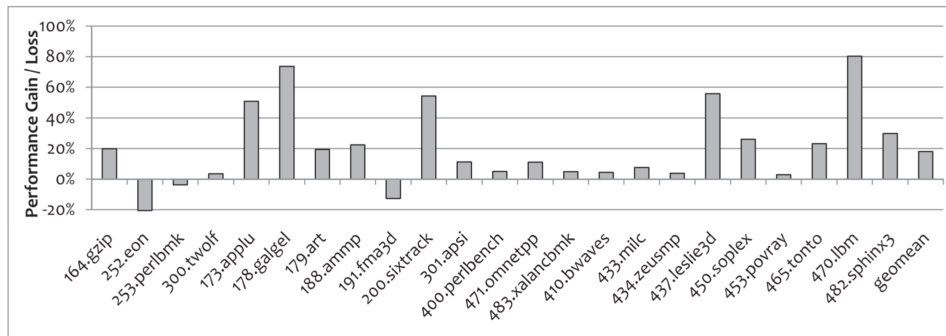

Fig. 12. Performance gain/loss with multiversion code in direct method.

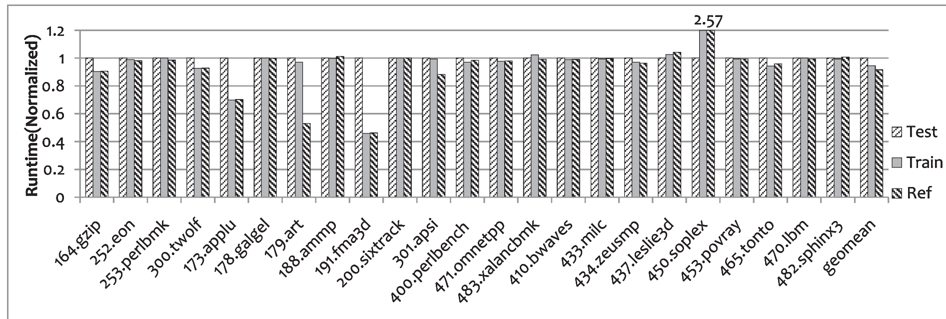

Fig. 13. Performance of static profiling with different input sizes of profiling *(baseline is the runtime while profiling with test input set)*.

as high as 80%. However, 252.eon shows a drop in performance. This is because the multiversion method would lead to significant code inflation and incur more I-cache misses. For 252.eon, the performance loss caused by extra I-cache misses overwhelms the performance gains from multiversion method.

6.6.3. Static Profiling. For the static profiling method, we have followed the profiling process with different input sets (one of *test*, *train* and *ref* input sets in SPEC) and evaluate the performance with all *ref* input sets. The result is shown in Figure 13.

For most benchmarks, profiling with smaller input sets yields lower performance, especially for 164.gzip and 179.art. We may expect better performance with profiles generated from with the *ref* input sets. However, one anomaly to report is that 450.soplex has its best performance with profiles from the *test* input set. The 450.soplex benchmark has multiple input datasets. We selected one of the *ref* sets for the profiling run. However, this profile is probably less representative than the test profile. Although we could combine multiple ref sets in the profiling process to eliminate this anomaly, this may miss the point that the effectiveness of static profiling is very much dependent on the quality of profiles. A larger input set does not guarantee a more representative profile, unless the input for profiling is exactly the input for actual execution.

Table III summarizes the number of MDAs which are not detected while profiling with different input sets. As we can see in Table III, when profiling with a larger input dataset, the number of unidentified MDA would reduce except for 252.eon and 450.soplex.

Table III. The Number of Unidentified MDAs while Profiling with Different Input Sets (only one set is selected if multiple datasets exist)

Benchmarks	Test	Train	Ref
164.gzip	46	46	0
252.eon	3.22E+09	3.22E+09	3.22E+09
253.perlbmk	74230241	75776	75776
300.twolf	2	0	0
173.applu	0	12	0
178.galgel	4504104	4930086	0
179.art	3.6E+09	3.6E+09	0
188.ammp	0	0	0
191.fma3d	2.63E+09	368432	0
200.sixtrack	0	0	0
301.apsi	2.94E+08	2.94E+08	0
400.perlbench	7166508	1244769	0
471.omnetpp	1.84E+08	48638638	0
483.xalancbmk	4	12761	0
410.bwaves	2.74E+08	0	0
433.milc	2592000	6	0
434.zeusmp	1.13E+09	644100	0
437.leslie3d	1	21168	0
450.soplex	24669824	4.03E+09	4.03E+09
453.povray	518	0	0
465.tonto	1.51E+09	262	0
470.lbm	0	0	0
482.sphinx3	162	0	0

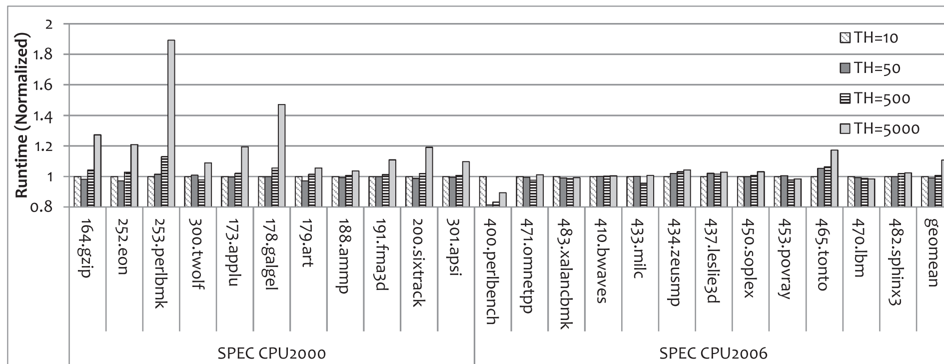

Fig. 14. Performance of dynamic profiling with different thresholds *(baseline is the runtime while TH = 10)*.

6.6.4. Dynamic Profiling. An appropriate heating threshold (i.e., a threshold to determine if a block is hot or not) in a two-phase binary translation system is critical to obtain a good overall performance. As discussed in Section 4.2, a high heating threshold can profile deeper into execution and uncover more potential MDAs, but at the cost of a much greater profiling overhead. To show the impact of different heating thresholds, we vary the value of thresholds from 10 up to 5000 (the threshold is a simple cumulative count over the whole execution). As shown in Figure 14, a threshold around 50 can strike a better balance and yield the best overall performance. Figure 15 gives the percentage of MDAs that can be detected by the dynamic profiling method with different thresholds. For most benchmarks, a threshold around 10 can detect most of MDAs in the application. However, for some benchmarks a threshold smaller than 50 is insufficient to uncover major MDA instructions, and hence will pay later when misaligned traps are encountered. For example, the 400.perlbench benchmark definitely

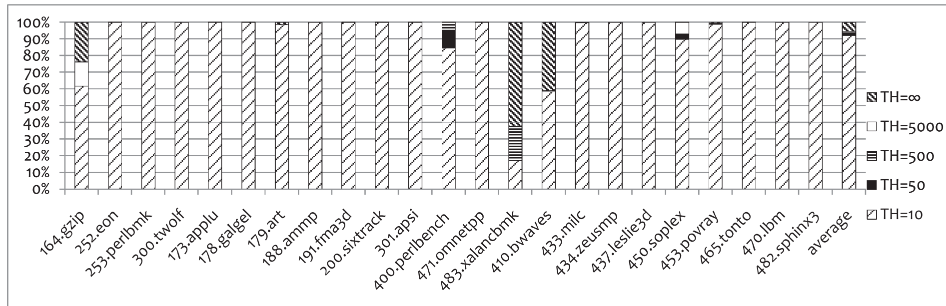

Fig. 15. Percentage of MDAs that can be detected by dynamic profiling method with different thresholds.

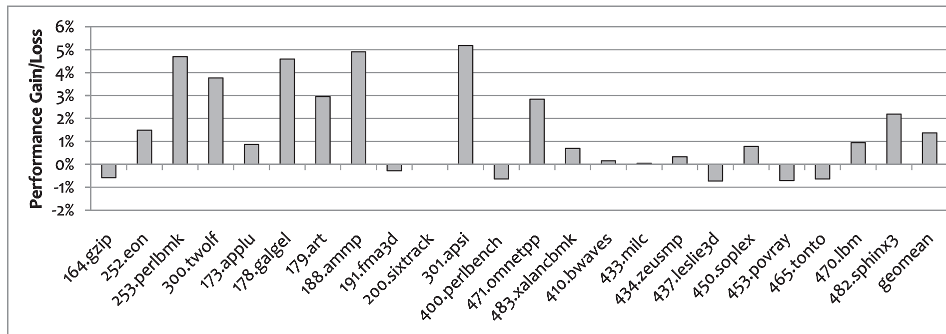

Fig. 16. Performance gain/loss with code rearrangement in exception handling method.

needs a threshold greater than 10. In general, a threshold greater than 500 offers no benefit.

Several programs, such as 164.gzip, 253.perlbmk, and 178.galgel, suffer the excessive profiling overhead with essentially no performance gain from the reduction of misalignment exceptions. We can also observe that the impact of this threshold for SPEC CPU2000 benchmarks is more significant than for SPEC CPU2006 benchmarks. This is because the input set size of SPEC CPU2000 is smaller than SPEC CPU2006, and a larger threshold could yield more profiling overhead for SPEC2000.

As shown in Figure 15, there are still a large amount of MDAs that cannot be detected with a threshold around 5000 in 483.xalancbmk and 410.bwaves.

6.6.5. Exception Handling. The exception handler-based method can be enhanced by code rearrangement optimization. Since misaligned traps will trigger the generation of MDA code sequence, multiple patches to the same basic block decreases code locality. If we can reposition the newly generated MDA code, some programs can be sped up. As shown in Figure 16, code rearrangement can speed up 253.perlbmk, 178.galgel, 188.ammp and 301.apsi by 4–5%. However, the overall performance gain from repositioning the MDA code has only marginal performance impact (1.4% gain on average).

6.6.6. DPEH Method. When the initial dynamic profiling failed to catch frequent MDAs, or when the program incurred behavior changes, the translated code could be ineffective. Although our exception handler-based dynamic translation will continuously capture new MDAs undetected by the initial profiling and translate them into MDA code sequences, the cost of exception handling and later code repositioning could be a burden. When such cases are detected, simply discarding the translated code of a block and retranslating it may work better. Figure 17 (the baseline is DPEH method) shows

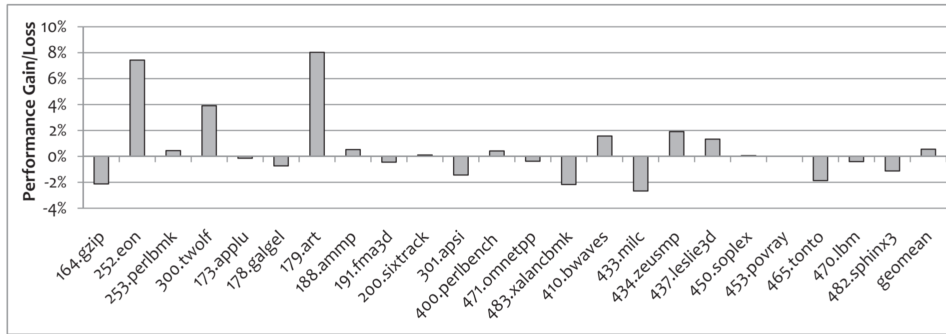

Fig. 17. Performance gain/loss with retranslation in DPEH method.

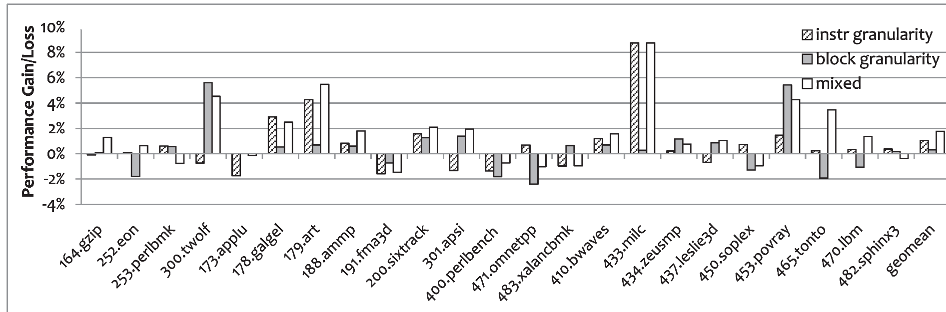

Fig. 18. Performance gain/loss with multiversion code in DPEH method *(baseline is the runtime of DPEH method)*.

the results of retranslating a basic block if four misalignment exceptions are raised in the translated code of the block at runtime. The data shows that some benchmarks benefit significantly from retranslation while some other benchmarks degrade slightly. Overall, the benefit of retranslation is not substantial.

6.6.7. Multiversion Code. In Section 6.6.2, we discussed the performance benefit of multiversion code on the direct method. In this section, we will discuss the performance impact of generating multiversion code for identified MDA instructions.

To evaluate the benefits of multiversion code method, we first implement this optimization based on the DPEH method. We generate multiversion code according to the profile information which we collected in dynamic profiling stage; and for MDAs that occurred in exception handling stage we consider the memory access instruction is likely to have behavior changes, so we also generate the multiversion code for that instruction. For programs with changing memory reference behavior, generating multiversion code might work best. However, it would be interesting to know which granularity to generate multiversion code would work best.

Figure 18 compares the execution time while generating multiversion code on instruction granularity, block granularity, and mixed granularity. With block granularity, we do not generate multiversion code for a single MDA instruction. When on mixed granularity, we generate multiversion code for MDA instructions that follow the same address pattern on block granularity and generate multiversion code for other MDA instructions on instruction granularity.

The data shows that multiversion code method provides little improvement on average for the DPEH method. This is because the data addresses of MDA instructions

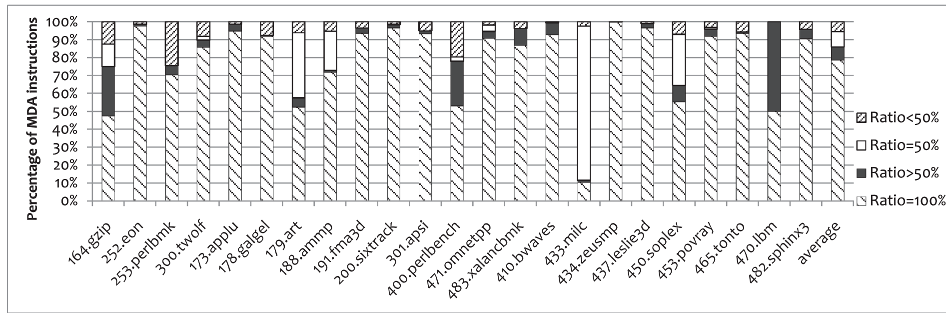

Fig. 19. Percentage of MDA instructions classified by misaligned ratio. Ratio = Number of MDAs of a MDA instruction/Number of memory references of the MDA instruction.

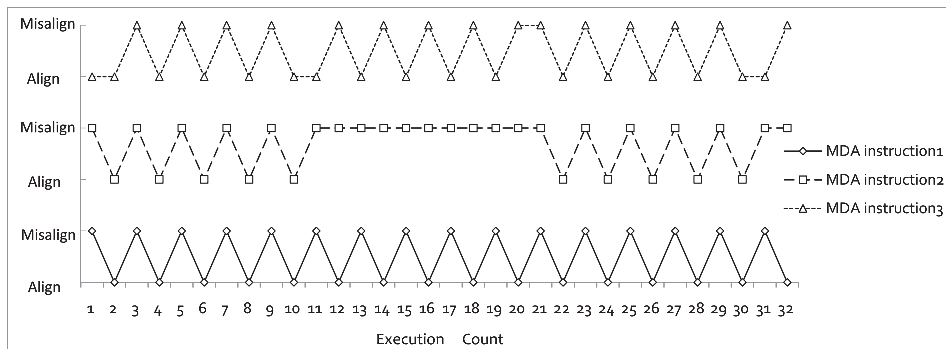

Fig. 20. The data access patterns of 3 MDA instructions in the same basic block of 433.milc.

in the benchmark suite are biased toward misalignment most of the time. In such a case, as long as we can efficiently identify MDA candidates and generate MDA code sequences for them, the performance should be good. Figure 19 gives the percentage of MDA instructions that are always misaligned, frequently misaligned, and frequently aligned. As we can see in the figure, only about 5% MDA instructions are frequently aligned. Besides, the multiversion method incurs overhead at runtime since the alignment checking instructions also consume cycles.

In Section 5.1, we discussed that generating multiversion code on basic block granularity can decrease runtime overhead. Nevertheless, as Figure 18 shows, multiversion code on block granularity adds very little benefit. Figure 20 shows the data access patterns of 3 different MDA instructions in the same basic block. As the figure shows, the data access behavior of some MDA instructions frequently changes, and different MDA instructions in the same basic block may exhibit different data access behaviors. We can also observe that the efficiency of generating multiversion code on basic block granularity is closely related to the profiling threshold. As the figure shows, if the profiling threshold is less than 10, we may incorrectly assume the three MDA instructions follow the same data access pattern.

Figure 21 shows the performance gain/loss from generating multiversion code based on the static profiling method. As the figure shows, the performance is similar to the DPEH method; the overall performance gain is miniscule (about 1.2%), except for a few cases where 5–8% of performance may be gained.

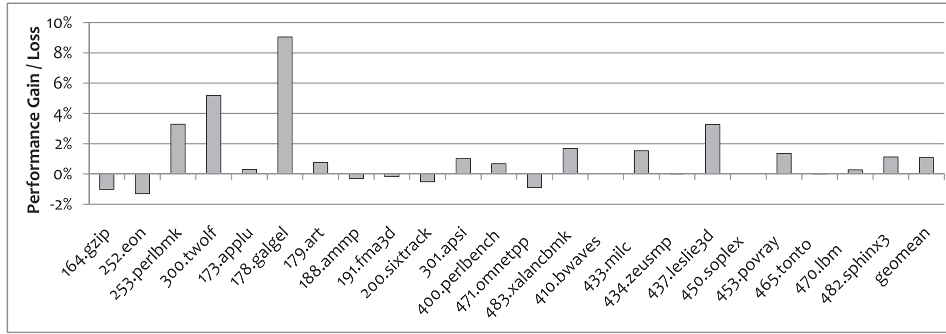

Fig. 21. Performance gain/loss with multiversion code in direct method and static profiling method.

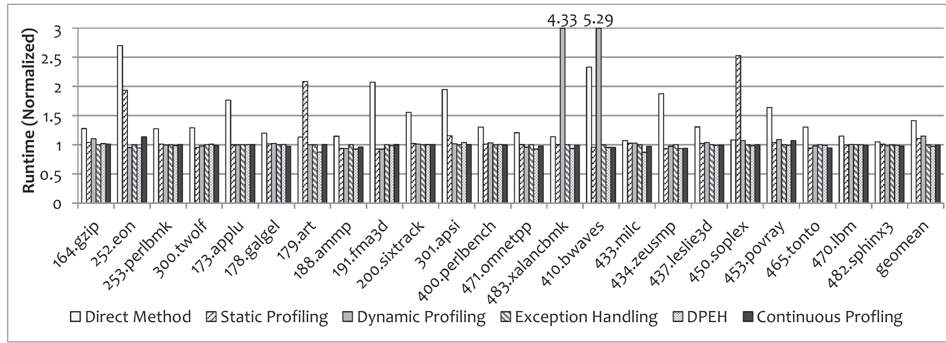

Fig. 22. Performance for different MDA handling mechanisms *(baseline is the runtime of exception handling method)*.

6.7. Overall Comparison

Now that we have explored the design and parameter space for the direct method, static profiling, dynamic profiling, exception handling, DPEH, and continuous Profiling method, we can compare them to see which one is more effective on handling MDAs. For this evaluation, we compare the execution time of the benchmarks with different MDA handling mechanisms. Each mechanism is configured to achieve the best performance (for static profiling method, we get the profile data with train input set; and for dynamic profiling method, we set the heating threshold to 50). The result is shown in Figure 22. As shown in the figure, exception handling method has already achieved higher performance than other MDA handling mechanisms: 17% better than dynamic profiling, 13% better than static profiling, and 44% better than direct method on average. However, DPEH and continuous profiling methods outperform exception handling method by 2.8% and 1%, respectively.

Comparing Figure 22 to Figure 10, we can see that the measured runtime performance tracks our estimated MDA overhead very well. For example, the significant slowdown of dynamic profiling on 483.xalancbmk and 410.waves and static profiling on 450.soplex in Figure 26 is due to MDA overhead which was already reflected in Figure 12. Although some individual programs may still have a large room for improvement (such as 164.gzip and 178.galgel), our approaches can be most effective for the remaining programs in the benchmark suite.

The continuous profiling method is a variation of the exception handling method. It uses the exception handler to collect MDA profiles instead of using interpretation or

Table IV. The Number of MDAs that Cannot be Detected by the Dynamic Profiling Mechanism (heating threshold = 50)

Benchmarks	Num of MDAs	Benchmarks	Num of MDAs
164.gzip	1.56E+08	471.omnetpp	38979
252.eon	24630	483.xalancbmk	8.32E+09
253.perlbmk	0	410.bwaves	4.15E+10
300.twolf	0	433.milc	1.34E+08
173.applu	1716	434.zeusmp	1716
178.galgel	3436	437.leslie3d	1716
179.art	3.12E+08	450.soplex	9.33E+08
188.ammp	0	453.povray	2.41E+08
191.fma3d	321789	465.tonto	116450
200.sixtrack	235950	470.lbm	0
301.apsi	1908	482.sphinx3	1
400.perlbench	57874640		

binary instrumentation. The data shows that the continuous profiling method works slightly worse than the DPEH method. This is because the accumulated MDA profile may incur extra overhead (e.g., some instructions are still translated to MDA code sequence, although they do not have MDA with the current input).

As discussed in Section 4, for applications that have frequent memory reference behavior changes, the dynamic profiling method may not work very well. As shown in Table IV, when the hot threshold is 50, there are still a large number of MDAs which can't be detected by the dynamic profiling method in 164.gzip, 179.art, 483.xalancbmk, 410.bwaves, 433.milc 450.soplex, and 453.povray. As we can see in Figure 22, these seven benchmarks are exactly the applications that suffer significant performance degradation compared with DPEH mechanism (8% for 164.gzip, 14% for 179.art, 340% for 483.xalancbmk, 433% for 410.bwaves, 15% for 433.milc 8% for 450.soplex, and 9% for 453.povray).

For the static profiling mechanism, the performance of most benchmarks is similar to DPEH mechanism. Table III summarizes the number of MDAs which are not detected while profiling with train input set. As we can see in Table III, while profiling with train input set, there are still a large number of MDAs in 252.eon, 179.art, and 450.soplex. The performance of these benchmarks has a significant degradation when compared with DPEH (91% for 252.eon, 13% for 179.art, and 155% for 450.soplex).

The direct method mechanism is generally worse than all others, simply because it indiscriminately increases the instruction overhead for all nonbyte memory instructions.

6.8. Evaluation of Full-Fledged Adaptation Method

In Section 5.2, we discussed how a full-fledged adaptive method can be implemented, and we speculated that the implementation may not worth pursuing. To verify our speculation, we have implemented the fully adaptive method (as shown in Figure 7) in our system. The performance is shown in Figure 23 (with results normalized to the execution time of the exception handling method). As shown in Figure 23, the full-fledged adaptive method incurs significant runtime overhead. Only a few benchmarks do better with it while most other benchmarks suffer significant performance degradation. As shown in Figure 19, about 80% of MDA instructions are always misaligned, and only about 5% MDA instructions are frequently aligned. Therefore, less than 20% of MDA instructions exhibit a phased behavior of the type [MDA $\rightarrow$ non-MDA]. If we want to convert the MDA sequence back to normal memory operation, the control must be redirected to the BT monitor and one context switch is needed. Therefore, only when the number of consecutive aligned memory accesses of an MDA instruction exceeds a threshold, the MDA code sequence is converted back (in our system we set the

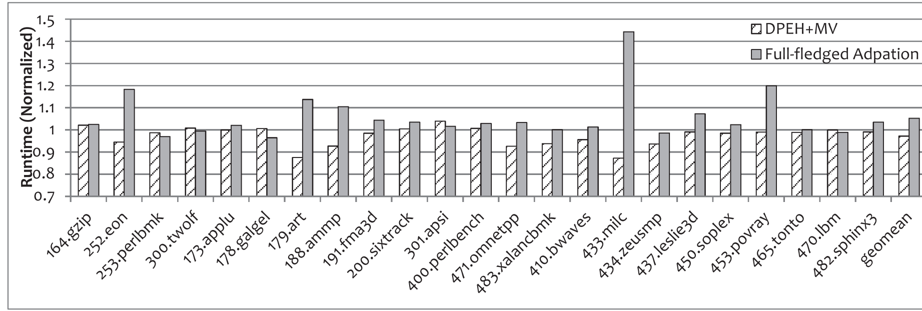

Fig. 23. Performance of full-fledged adaptation in comparison with the DPEH method.

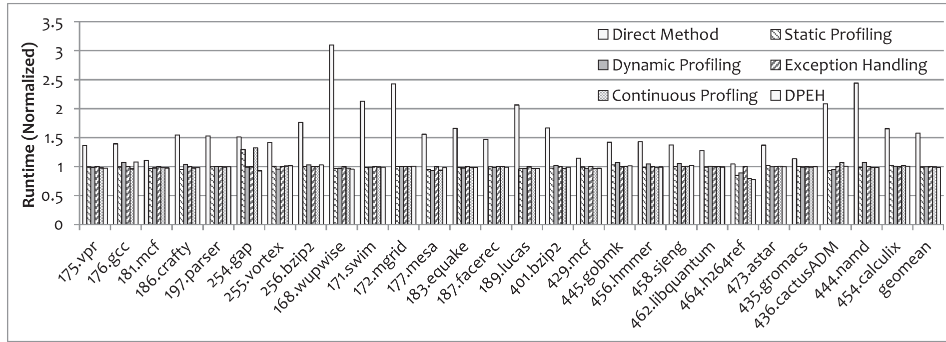

Fig. 24. Performance of different MDA handling mechanisms on no-MDA benchmarks *(baseline is the runtime of exception handling method)*.

threshold to 1000 because the overhead of context switch is high). As a result, only a few MDA instructions can benefit from the full-fledged adaptive method while the majority of MDA instructions suffer high profiling overhead.

6.9. Impact on MDA-Free Benchmarks

Up to now, we have studied the impact of different MDA handling mechanisms on programs that have a significant number of MDAs. However, for programs that have no MDAs, what is the performance impact of the MDA handling mechanisms? Figure 24 shows the performance impact of MDA handling mechanisms on programs where MDAs are rare or nonexistent (the baseline is the performance without MDA handling mechanisms). As shown in the figure, for most benchmarks the performance of DPEH, dynamic profiling and static profiling methods are basically the same as the baseline. Direct method shows severe performance degradation because it introduces a large number of redundant instructions from unnecessary MDA code sequence.

7. RELATED WORK

Most binary translation systems which translate the X86 ISA have dealt with MDA instruction handling.

QEMU [Bellard 2005] is a portable processor emulator that relies on dynamic binary translation to achieve a reasonable emulation speed. It handles MDA by directly translating all multibyte memory operations into MDA code sequences.

FX!32 [Chernoff et al. 1998; Hookway and Herdeg 1997] is a software emulator that allows X86 programs to run on DEC Alpha-based systems on Windows NT. Unlike

other X86 dynamic binary translation systems, it does not dynamically translate code. Rather, it performs translation and optimization between program runs. Based on the profiles collected from previous runs, the translator and the optimizer can decide which memory operations are good candidates of MDA, and such instructions are translated into the special MDA code sequence as shown in Figure 2. Since the profiles used in FX!32 were collected from previous runs, not from the current run, we consider it as static profiling. Although the approach used in FX!32 was quite effective, MDA exceptions can still occur frequently for programs that exhibit different memory reference behavior on runs with varying input datasets.

The IA-32 EL [Baraz et al. 2003] is also a software emulator that improves performance of IA-32 applications on the IA-64-based systems. It is a two-phased dynamic binary translator. In the first phase, source binary is translated into native code with instrumentations to identify hotspots and collect profiles. In the second phase, hotspots are retranslated and optimized using collected profiles. The IA-32 EL handles MDA in three stages: (1) In the first phase, all instructions that may have MDA are lightly instrumented; (2) if an MDA is detected, the block containing the MDA instruction is retranslated, and that MDA instruction is heavily instrumented to provide more detailed misalignment information; and (3) in the second phase, the MDA instructions detected in the first phase are translated to MDA code sequences. To cope with MDAs that appear only after the optimization phase, the instructions which are empirically considered to have danger of incurring MDA later on are instrumented to detect MDA in the hot code. However, the published work does not reveal how to identify a block where memory operations have a significant danger of MDA.

Although the Transmeta Code Morphing System [Dehnert et al. 2003; Coon et al. 2006] also dynamically translates x86 binary, it detects and avoids MDA using hardware solutions. UQDBT [Ung and Cifuentes 2000] is another dynamic binary translation system which migrates x86 applications to SPARC machines. The SPARC processors require natural alignments, but there were no discussions on MDA handling in the published work.

8. SUMMARY AND CONCLUSIONS

Binary translation has been used widely in various applications such as ISA migration, runtime code inspection and optimization, and dynamic instrumentation. A critical but underinvestigated issue is how to efficiently and effectively handle misaligned data accesses. Existing techniques either incur excessive overhead or cannot adapt to the change of memory access behavior at runtime.

In this article, we have studied the strength and weakness of many existing MDA handling techniques. The direct method which translates every nonbyte memory operation into the MDA code sequence is naive and incurs excessive overhead. Using static profile feedback may fail to catch those MDA candidates that do not show up with the training runs. Using dynamic profiling is less sensitive to profile selection and may fail to catch important MDA candidates during the profiling phase.

We have proposed an exception handler-based mechanism which can be considered a lightweight continuous profiling of MDA operations. As long as memory operations are executed without misaligned exceptions, they can run at full speed without interference. When a misaligned exception occurs, the exception handler will translate this particular memory instruction into the MDA code sequence. Based on this lightweight profiling approach, we combine it with dynamic profiling which can effectively catch many MDA candidates during the typical profiling phase. Handling many MDA candidates at the same time can avoid the cost of context switching between native execution and the binary translation. We also make our method more adaptive by dynamically translating those memory operations that change between MDA and aligned accesses

into two-version code. Furthermore, this method is improved by code repositioning and retranslation optimizations to enhance instruction spatial locality. We have shown that our enhanced methods are more adaptive to program behavior change than all existing methods, as it outperforms the static profiling method by 13%, the dynamic profiling method by 17%, and the direct method by 44%, on 23 SPEC2000 and SPEC2006 benchmark programs with frequent misaligned data accesses.

REFERENCES

ADAMS, K. AND AGESEN, O. 2006. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*. ACM, New York, 2–13.

BALA, V., DUESTERWALD, E., AND BANERJIA, S. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00)*. 1–12.

BARAZ, L., DEVOR, T., ETZION, O., GOLDENBERG, S., SKALETSKY, A., WANG, Y., AND ZEMACH, Y. 2003. Ia-32 execution layer: A two-phase dynamic translator designed to support ia-32 applications on itanium®-based systems. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO36)*. IEEE Computer Society, Los Alamitos, CA, 191.

BELLARD, F. 2005. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC'05)*. USENIX Association, Berkeley, CA, 41–46.

BRUENING, D., GARNETT, T., AND AMARASINGHE, S. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. IEEE Computer Society, 265–275.

BUNGALE, P. P. AND LUK, C.-K. 2007. Pinos: A programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE'07)*. ACM, New York, 137–147.

CHERNOFF, A., HERDEG, M., HOOKWAY, R., REEVE, C., RUBIN, N., TYE, T., YADAVALLI, S. B., AND YATES, J. 1998. Fx!32: A profile-directed binary translator. *IEEE Micro 18,* 2, 56–64.

COMPAQ. 2002. *Alpha Architecture Reference Manual* 4th Ed. Compaq.

COON, B., DSOUZA, G., AND SERRIS, P. 2006. Patent: Pipeline replay support for unaligned memory operations. United States patent, 7134001B1.

DEHNERT, J. C., GRANT, B. K., BANNING, J. P., JOHNSON, R., KISTLER, T., KLAIBER, A., AND MATTSON, J. 2003. The transmeta code morphing™software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. IEEE Computer Society, Los Alamitos, CA, 15–24.

DRONGOWSKI, P. J., HUNTER, D., FAYYAZI, M., KAELI, D., AND CASMIRA, J. 1999. Studying the performance of the fx!32 binary translation system. In *Proceedings of the 1st Workshop on Binary Translation*. ACM, New York, 15–24.

DUESTERWALD, E. 2005. Design and engineering of a dynamic binary optimizer. *Proc. IEEE 93,* 2, 436 –448.

EBCIOĞLU, K. AND ALTMAN, E. R. 1997. Daisy: Dynamic compilation for 100. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*. ACM, New York, 26–37.

HAZELWOOD, K. AND SMITH, M. D. 2006. Managing bounded code caches in dynamic binary optimization systems. *ACM Trans. Archit. Code Optim. 3,* 3, 263–294.

HENNESSY, J. L. AND PATTERSON, D. A. 2006. *Computer Architecture: A Quantitative Approach* 4th Ed. Morgan Kaufmann, Elsevier

HOOKWAY, R. J. AND HERDEG, M. A. 1997. Digital fx!32: Combining emulation and binary translation. *Digital Tech. J. 9,* 1, 3–12.

INTEL. 2009. *Intel 64 and IA-32 Architectures Software Developers Manual* Volume 1-3. Intel Corporation.

KE CHEN, W., LERNER, S., CHAIKEN, R., AND GILLIES, D. M. 2000. Mojo: A dynamic optimization system. In *Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*. ACM, New York, 81–90.

NETHERCOTE, N. AND SEWARD, J. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. 89–100.

PACKIRISAMY, V., WANG, S., ZHAI, A., CHUNG HSU, W., AND CHUNG YEW, P. 2006. Supporting speculative multithreading on simultaneous multithreaded processors. In *Proceedings of the 13th International Conference on High Performance Computing*. Springer, Berlin, 148–158.

QIN, F., WANG, C., LI, Z., KIM, H.-S., ZHOU, Y., AND WU, Y. 2006. Lift: A low-overhead practical information flow tracking system for detecting security attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO39)*. IEEE Computer Society, Los Alamitos, CA, 135–148.

SCOTT, K., KUMAR, N., VELUSAMY, S., CHILDERS, B., DAVIDSON, J. W., AND SOFFA, M. L. 2003. Retargetable and reconfigurable software dynamic translation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'03)*. IEEE Computer Society, Los Alomitos, CA, 36–47.

SITES, R. L., CHERNOFF, A., KIRK, M. B., MARKS, M. P., AND ROBINSON, S. G. 1993. Binary translation. *Comm. ACM 36,* 2, 69–81.

SMITH, J. E. AND NAIR, R. 2005. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, Elsevier.

TRANSITIVE TECHNOLOGIES LTD. 2001. http://www.transitive.com/products/solsparc_solx86.

UNG, D. AND CIFUENTES, C. 2000. Machine-Adaptable dynamic binary translation. *SIGPLAN Not. 35,* 7, 41–51.

Received February 2010; revised November 2010; accepted February 2011