2009 International Symposium on Code Generation and Optimization

An Evaluation of Misaligned Data Access Handling Mechanisms in Dynamic Binary Translation Systems

Jianjun Li, Chenggang Wu
Key Laboratory of Computer System and Architecture
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
{lijianjun, wucg}@ict.ac.cn

Wei-Chung Hsu
Department of Computer Science
University of Minnesota
Minnesota, USA
hsu@cs.umn.edu

***Abstract*—Binary translation (BT) has been an important approach to migrate application software across instruction set architectures (ISAs). Some architectures, such as X86, allow misaligned data accesses (MDAs), while most modern architectures have the alignment restriction that requires data to be aligned in memory on natural boundaries. In a binary translation system, where the source ISA allows MDA and the target ISA does not, memory operations must be carefully translated to satisfy the alignment restriction. Naive translation will cause frequent misaligned data access traps to occur at runtime on the target machine, and severely slow down the migrated application.**

This paper evaluates different approaches in handling MDA in binary translation systems. It also proposes a new mechanism to deal with MDAs. Measurements based on SPEC CPU2000 and CPU2006 benchmark show that the proposed approach can significantly outperform existing methods.

***Keywords*-optimization; misaligned memory access; binary translation**

I. Introduction

Binary Translation (BT) [1] is a technique used to migrate application binaries from one ISA to another [2] [3] [4]. When the target ISA is the same as the source ISA, BT can have multiple interesting applications such as dynamic optimization [5] [6], dynamic speculation and parallelization [7], dynamic instrumentation [8] [9], and dynamic software security enforcement [10]. When the target ISA is different from the source ISA, dynamic BT is the key technology for implementing process virtual machines [11].

Some architectures do not have alignment restrictions [12] where data operands must start at addresses that are multiples of the data's natural size. 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 to handle the required memory accesses. For architectures that disallow MDA, such as MIPS, ALPHA, IA-64, and most modern RISCs, misaligned data accesses will always generate misaligned access traps. Since the cost of handling a misalignment exception is very high, frequent MDAs will severely slow down the execution of the application. This is usually not a problem because the compilers for modern RISCs ensure data are properly allocated to satisfy the alignment restriction. 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 execution of the translated memory operations is likely to generate MDA. Therefore, in a binary translation system, when the source architecture allows MDA, memory operations must be translated carefully to minimize the performance impact from MDA.

Different mechanisms to deal with MDA have been implemented in existing binary translation systems, such as Code Morphing Software [13], QEMU [4], FX!32 [2] and IA-32 EL [3]. Code Morphing Software handles MDA with the assistance of dedicated hardware. QEMU translates any possible MDA operation into a sequence of byte operations to avoid MDA, but this approach incurs significant instruction overhead. FX!32 and IA-32 EL can selectively translate problematic MDA memory operations into byte operations or some special code sequences to minimize overhead, but their approaches are not adaptive to behavior changes in programs. For programs exhibit different MDA behavior under different input data sets or as part of phase changes, frequent MDA traps can still occur in FX!32 and IA-32 EL.

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

- A comprehensive study of MDA handling mechanisms in BT systems for the X86 architecture, including a direct method, a static profiling method and a dynamic profiling method.
- A new mechanism, which can adaptively handle MDA according to behavior changes of the program, is proposed. For some applications, the performance gain can be more than 10% when compared with existing

Table I: MDAs in SPEC CPU2000 and CPU2006

Benchmarks	NMI	Number of MDAs	Ratio	Benchmarks	NMI	Number 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	3,825	67,272,361,837	12.09%
173.applu	306	2,243,041,896	1.60%	434.zeusmp	3,484	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	3,304	31,740,862	0.00%
200.sixtrack	1324	8,673,947,498	4.21%	465.tonto	1,748	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%				

mechanisms.

The rest of this paper is organized as follows. Section II gives a brief overview of MDAs in BT system and Section III describes basic MDA handling methods. Section IV extends basic MDA handling with the assistance of exception handling to catch new MDAs on-the-fly. It also introduces a more adaptive mechanism to handle MDA based on the use of misalignment exception handler. Section V describes the experimental framework in which the misalignment handling mechanisms are evaluated and Section VI presents the experimental results. Section VII summarizes and concludes this work.

II. MDAs in BT Systems

How often would MDA occur in a BT system if the source ISA has no alignment restrictions? Let us consider a case where the source ISA is X86 and the target ISA is Alpha. Table I lists the number of MDAs encountered for the SPEC CPU2000 and SPEC CPU2006 benchmarks (with *ref* input set) which are compiled for X86 by the pathscale2.4 compiler on an X86/Linux system. In the table, NMI stands for the number of instructions that references misaligned data, and Ratio is the number of MDAs divided by the total number of memory accesses. Table I shows that some programs have very frequent MDAs (e.g. 410.bwaves with 12.67% of all memory operations are MDAs, and 433.milc, 12.09%) 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 [15] [16], then the average overhead could be as high as 9.5K seconds on a 1GHz machine.

Conventional wisdom has it that even in computers that allow misaligned data access, programs compiled with aligned accesses would run faster [12]. Therefore, 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 icc, the pathscale, and the gcc compiler. Nevertheless, this optimization is not 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 a set of SPEC benchmarks. Figure 1 shows the performance with alignment optimization (using the *pathscale* and the *icc* compilers) on X86 machines. There were no significant performance advantages with data alignment (1% for *pathscale* and 1.8% for *icc*, 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 complied with data aligned.

In addition, we have observed frequent MDAs in the X86 shared libraries, such as libc.so.6, and libgfrotran.so.6. We have noticed that more than 90% of MDAs occurred

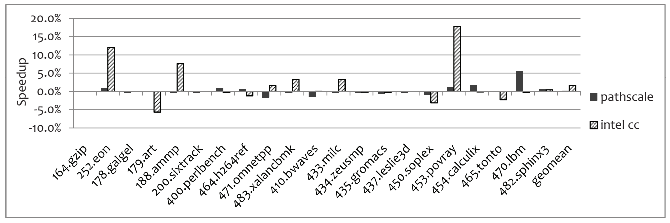
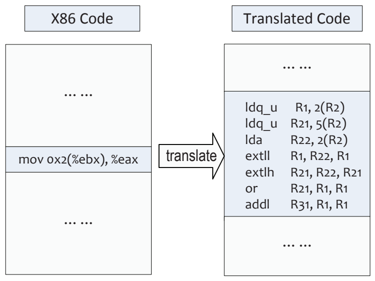


Figure 1: Performance with alignment optimization flags for pathscale and intel cc



```
mov 0x2(%ebx), %eax
```

```
ldq_u   R1, 2(R2)
ldq_u   R21, 5(R2)
lda     R22, 2(R2)
extll   R1, R22, R1
extlh   R21, R22, R21
or      R21, R1, R1
addl    R31, R1, R1
```

Figure 2: MDA Handling with direct method on Alpha

in 164.gzip, 400.perlbench, and 483.xalancbmk are actually come from shared libraries. Even if some ISVs release their binaries with data alignment enforced, as long as the application uses 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.

III. Direct Translation and Profiling Based Methods

Several different MDA handling mechanisms have been implemented in existing binary translation systems. The mechanisms can be classified into four categories: direct method(Section III-A), static profiling based (Section III-B), dynamic profiling based (Section III-C), and exception handling based (Section IV).

In current binary translation systems, QEMU [4] use direct method to handle MDAs, FX!32 [2] uses a method similar to static profiling, and the dynamic profiling method is introduced by IA-32 EL [3]. These are all pure software techniques for handling MDA. Hardware technique is employed in Transmeta's Code Morphing Software [13] [14], and it is beyond the scope of this paper.

A. Direct Method

For any data access that is greater than a byte, a misaligned data accesses could happen and would cause misaligned access traps to be generated on machines with alignment requirements. Some architectures without hardware support to MDAs usually provide a code sequence (we call it MDA code sequence in this paper) to access misalignment address without triggering misalignment exceptions. In fact, the misalignment exception handler can use the same MDA code sequence to handle the misaligned data access. The direct method simply translates all memory access instructions into the MDA code sequence [4]. 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[1]. Figure 2 illustrates the MDA code sequence on Alpha [20] [21]. 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. 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.

It is obvious that executing the MDA code sequence would be much faster than going through an exception handling. Hence, direct translation of memory operations to MDA code sequence can be faster if the memory operation is an MDA. However, the MDA code sequence is much slower, if compared to a single memory operation. Therefore when there are no misaligned data accesses, using the MDA code sequence would incur significant overhead. As a result if the 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 should remain as a regular memory operation, and leave the misaligned access (suppose to be rare) to exception handling. This is the basic idea of MDA handler. The challenging issues here are a) how to determine the probability of the memory operation being an MDA, b) is the behavior of the MDA operation stable, and c) if the behavior tends to change, is it changing with a predictable pattern.

B. Static Profiling Based Method

Some BT systems rely on static profiling to identify memory operations that are likely to be MDA. A profiling run with training input set collects information on MDA. Guided by the profile, the binary translator generates the MDA code sequence for the memory operation with high MDA probability [2]. Figure 3 depicts the mechanism of the static profiling based method.

[1]*On MIPS and Itanium architectures, there are also several special instructions to support MDAs, for example, ldl/ldr on MIPS and shrp on Itanium. However, on some architectures, there are no special instructions to support MDA, so that the misaligned data access has to be handled in a sequence of byte or halfword memory operations.*

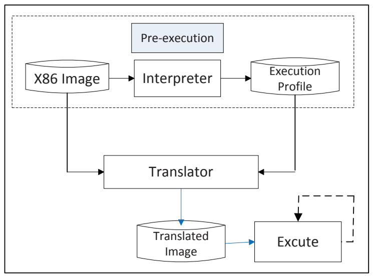


Figure 3: Static Profiling Mechanism

This mechanism is effective if the training input is representative and can reflect the real execution behavior of the program. However, this is not true in many cases. Additionally, many programs allocate data dynamically. On subsequent runs, a dynamically allocated data may or may not be aligned. Therefore, with static profiling, MDAs may still occur frequently. If the MDAs happen to be in a hot loop, the performance penalty could be very high.

C. Dynamic Profiling Based Method

In order to overcome the limitation of static profiling, many dynamic binary translation systems adopt a two-phase translation approach. In the first phase, the binary is either interpreted or directly translated without any optimization and profiles are collected from execution; in the second phase, hot regions identified from the first phase are retranslated and further optimized.

The dynamic profiling method is based on the two-phase binary translation approach. As depicted in the left hand 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 (the address of instruction which have misaligned access and the type of misalignment) is recorded. 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 [3]. In the experiments which we show in Section VI, 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 is adaptive to behavior changes with different inputs, and it does not require a separate profiling process. However, if the behavior change occurs after the code is generated, 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 capture 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

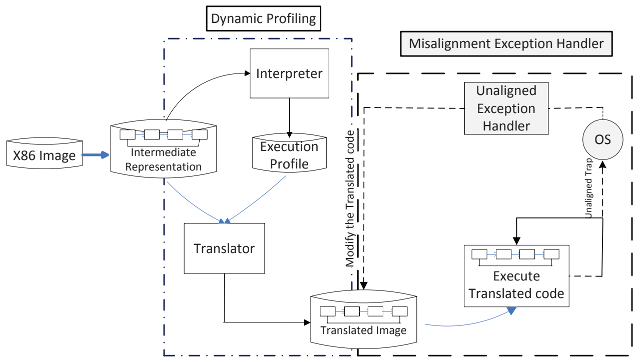


Figure 4: Dynamic Profiling & Exception Handling Mechanisms in MDA Handling

should be as high as 266K if we want to eliminate most MDAs. We definitely need more effectively ways to capture those MDA candidates.

IV. Exception Handling Based Method

Section III deals with the basic MDA handling methods used in existing BT systems. The basic handling methods attempt to minimize misaligned traps at runtime by translating memory operations into MDA code sequences. Execution profiles are used to selectively translate only the memory operations with high likelihood of MDA into the MDA code sequence. In this section, we enhance basic MDA handling with the help of exception handling information. A more adaptive MDA handling mechanism is also proposed.

An exception handling based method is illustrated in the right hand 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 analyse 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 (this memory is usually called code cache by BT systems) to store the MDA code sequence.
- Patch the offending memory operation to a branch instruction jumping 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 performance impact of MDA, the instruction that incurs MDA was patched immediately as soon as the first MDA exception is encountered during execution.

Figure 5 shows the translated native code on Alpha when a 4-byte load incurs misalignment exception at run-time [20].

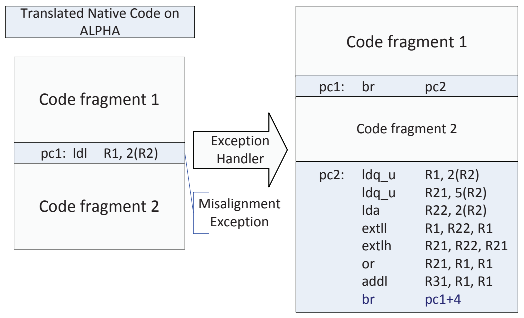


Figure 5: Example of the Exception Handling Based Method

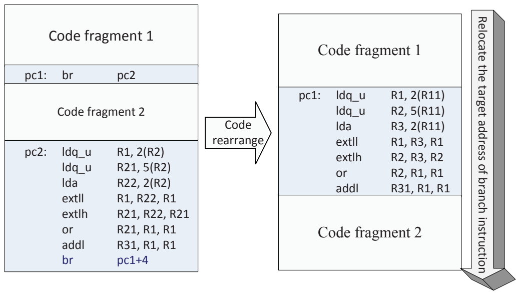


Figure 6: Example of Code Rearrangement Method

The code modified by the misalignment exception handler is given in the right hand 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 handling, 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.

A. Code rearrangement

One drawback of the exception handling based method is that the code locality is decreased after patching the memory operation to a jump. This may lead to increased instruction cache misses and result in significant performance loss. Code rearrangement can be used to get back the desired spatial locality. An example of code rearrangement (often called code repositioning) that we adopted is illustrated in Figure 6.

B. Combining Dynamic Profiling with Exception Handling (DPEH)

Although code rearrangement can avoid the loss of code locality from Exception Handling based method, it requires code relocation. If the number of memory operations incurring MDA is large, the cost of performing relocation may be excessive and unacceptable. One compromise is to combine the advantages of dynamic profiling with exception handling based method. Dynamic profiling is adopted to identify those memory operations with frequent MDA initially. At this stage, the dynamic profiling is operating with a relatively low threshold to minimize profiling overhead. This phase can identify many, if not most, memory operations with possible MDA. The remaining memory operations with MDA will be left for the exception handler to take care. In this way, the impact of code locality will be minimized, the need for code rearrangement with relocation is decreased, and we can enjoy the benefit of dynamic profiling with a low threshold. This new approach is actually the system illustrated in Figure 4.

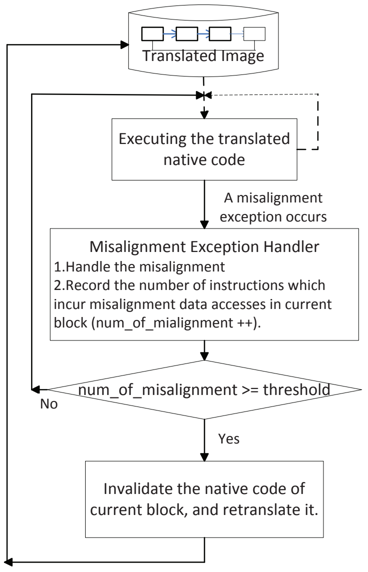
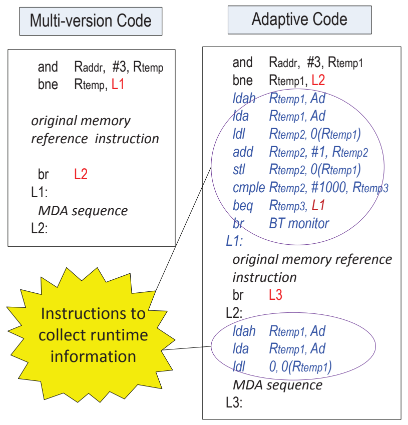

Translated Image
Executing the translated native code
A misalignment exception occurs
Misalignment Exception Handler
1.Handle the misalignment
2.Record the number of instructions which incur misalignment data accesses in current block (num_of_mialignment ++).
num_of_misalignment >= threshold
No
Yes
Invalidate the native code of current block, and retranslate it.

Figure 7: Overview of the retranslation process

C. Retranslation

Dynamic profiling can identify many frequent MDA operations in a block so that they can be turned into MDA code sequences at the first translation instead of being handled 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, and re-start the dynamic profiling and translation process for this block. The retranslation process is illustrated by the flowchart in Figure 7. 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 [5] except that Dynamo flush the entire code cache while our BT invalidates translated code at block granularity.



```
  and    Raddr,  #3, Rtemp
  bne    Rtemp, L1

 original memory
 reference  instruction

  br     L2
 L1:
  MDA sequence
 L2:
```

```
  and    Raddr,  #3, Rtemp1
  bne    Rtemp1, L2
  ldah   Rtemp1, Ad
  lda    Rtemp1, Ad
  ldl    Rtemp2, 0(Rtemp1)
  add    Rtemp2, #1, Rtemp2
  stl    Rtemp2, 0(Rtemp1)
  cmple  Rtemp2, #1000, Rtemp3
  beq    Rtemp3, L1
  br     BT monitor
 L1:
  original memory reference
  instruction
  br     L3
 L2:
  ldah   Rtemp1, Ad
  lda    Rtemp1, Ad
  ldl    0, 0(Rtemp1)
  MDA sequence
 L3:
```

Figure 8: An example of Adaptive Code method on Alpha architecture

D. Multi-version code

During the program execution, an instruction may access many different addresses, some of them aligned and some of them not. For this particular instruction, if aligned data addresses dominate the execution, translating them into the MDA code sequence would incur too much overhead. On the other hand, not translating them into the MDA code sequence would suffer from expensive exception handling. To solve this problem, we could generate two versions of native code for the code segment containing such instructions. One version translates the memory instruction into an MDA code sequence while the other version remains a single memory operation. An example is given on the left hand side of Figure 8. At run-time, the version to execute would be selected according to the actual memory address referenced.

The next option is on what granularity to generate the multi-version code. A simple approach is to generate two versions of codes for every memory operation. If the address is misaligned, the MDA code sequence is executed. Otherwise, the normal instruction is executed. This approach is more adaptive, in which it executes the MDA code sequence only when it is actually needed, but at the cost of some instruction overhead – the checking instructions do consume cycles. Perhaps a more efficient method could be devised. 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 multi-version code based on basic-block granularity.

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, but the original memory operation no longer have MDA? Should we convert the MDA code sequence back to normal memory operation? This method may be more effective than multi-version code method, but its implementation cost could be prohibitive. Figure 8 compares the multi-version code method and this truly adaptive method. As we can see in Figure 8, 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 replace the MDA code sequence back to original memory operation. Furthermore, even if we replace 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 above into account, we believe this seemingly more adaptive method may not be worth pursuing.

V. Evaluation environment

This section describes our test machine, the experiment framework, the benchmarks, the compilers, and the optimizations used in this experiment.

A. Machines

The mechanisms presented in Section III were evaluated on an one-processor Alpha ES40 machine running CentOS 4.2. The processor's on-chip cache hierarchy consists of a split L1 instruction and data caches and a unified L2 cache. The L1 caches are 2-way set associative and 64KB in size. The L2 cache is direct-mapped and 2MB in szie. The memory size of the machine is 4GB.

B. Experiment Framework

We have adopted the BT framework of DigitalBridge [18] [19] to evaluate all different MDA handling mechanisms. DigitalBridge, which is developed by our group, is a dynamic binary translator. It migrates X86 binaries to the Alpha architecture. Similar to FX!32 [2], DigitalBridge supports X86 binary codes to run on Alpha machines. Unlike FX!32, DigitalBridge uses on-line instead of off-line 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 [18]. Figure 9 shows the major components of DigitalBridge.

In this paper, all examples of MDA handling assume X86/Alpha as the guest/host machine model.

C. Benchmarks

The full suite of SPEC CPU2000 [22] and SPEC CPU2006 [23] benchmarks were used for evaluation in this paper. 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 according to Table I. The performance is represented as normalized ratios in our charts, therefore, we have also reported the geometric mean of the selected 21 benchmarks.

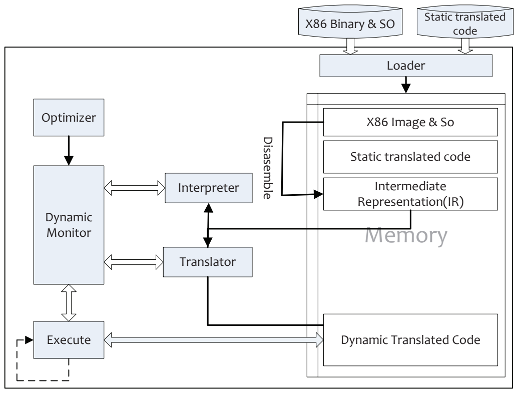


Figure 9: Framework of DigitalBridge

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

VI. Experimental Results

This section evaluates different MDA handling mechanisms discussed in Section III and Section IV. The execution time we show in this section is an average of three runs for each benchmark. Table II lists the parameters used for each MDA handling mechanism.

A. Dynamic Profiling

An appropriate heating threshold in a two-phase binary translation system is critical to obtain a good overall performance. As discussed in Section III, 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 10 (the baseline is TH=10), a threshold around 50 strikes a better balance and yields the best overall performance. 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 needs a threshold greater than 10. In general, a threshold greater than 500 offers little benefit. Several programs, such as 178.galgel, 164.gzip, 252.eon, 200.sixtrack, and 465.tonto, suffer from the excessive profiling overhead with essentially no performance gain from the reduction of misalignment exceptions.

B. Exception Handling

1) Code rearrangement: With the exception handling method, misaligned traps will trigger the generation of MDA code sequence. If we further reposition the newly generated MDA code, some programs can be sped up.

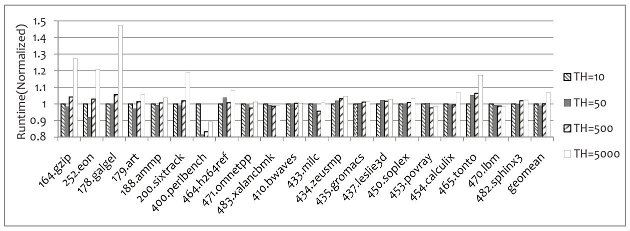


Figure 10: Performance with different thresholds

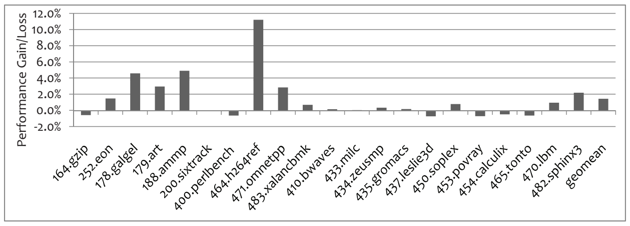


Figure 11: Performance gain/loss with code reaarangement

As shown in Figure 11, code rearrangement can speed up 464.h264ref by as much as 11%, and speed up 178.galgel and 188.ammp by 4-5%. However, the overall performance gain from repositioning the MDA code is marginal (only 1.5

2) DPEH method: In the DPEH method, dynamic profiling in the first phase will catch some MDA instructions. The remaining undetected MDA will be taken care of by the Exception Handling method. Compare the DPEH method to the Exception Handling method, as shown in Figure 12, three programs: 464.h264ref, 471.omnetpp, and 433.milc have more than 8% performance gain from the help of initial dynamic profiling. However, the overall gain is only about 2%. This indicates the simple Exception Handling method is working reasonably well.

3) Retranslation: 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 continously 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

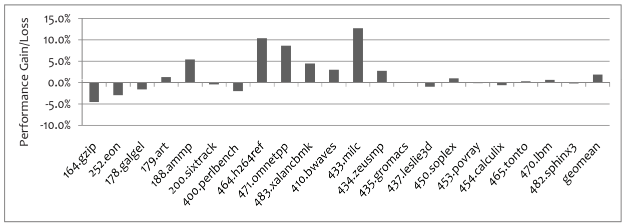


Figure 12: Performance gain/loss with dynamic profiling

Table II: MDA handling mechanisms and configuration choices

Mechanism	Configuration Choice	Description
Direct Method	*none*	none
Static Profiling	*none*	none
Dynamic Profiling	*Translation thresholds*	The translation threshold of the binary translation system.
Exception Handling	*Code rearrangement*	A method to reposition the MDA code generated by the Exception Handler.
Dynamic Profiling & Exception Hanlding	*Retranslation*	A method to retranslate a basic block when multiple MDA instructions have been detected by the Exception Handler.
	Multi-version code	Mechanism to selectively generate multi-version code.

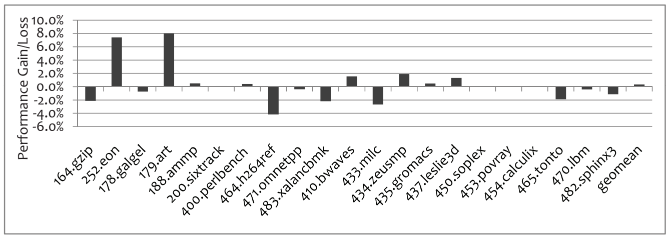


Figure 13: Performance gain/loss with Retranslation

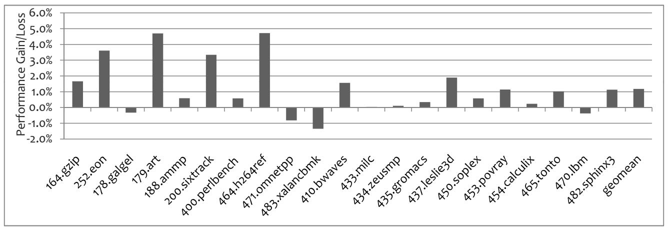


Figure 14: Performance gain/loss with Multi-version code

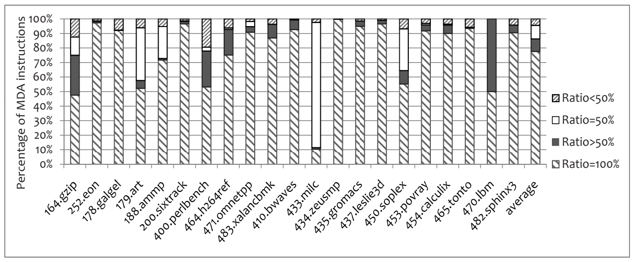


Figure 15: Percentage of MDA instructions classified by misaligned ratio. Ratio = Number of MDAs of a MDA instruction / Number of memory references of the MDA instruction

detected, simply discarding the translated code of a block, and retranslating it may work better. Figure 13 (the baseline is DPEH method) shows the results of retranslating a basic block if 4 misalignment exceptions are raised in the translated code of the block at run-time. The data shows that some benchmarks benefit significantly from retranslation, but some other benchmarks degrade slightly. Overall, the benefit of retranslation is not substantial.

4) Multi-version code: We generate multi-version code on top of the DPEH method, and the decision is based on the profile collected during the dynamic profiling stage and the exception handling stage. For programs with changing memory reference behavior, generating multi-version code might work best. Figure 14 (the baseline is DPEH method) compares the execution time with and without multi-version code method. However, the data shows that multi-version code method provides only marginal performance improvement, about 1.1% on average while some benchmarks see up to 4.7% of improvement.

This is because the data addresses of MDA instructions in the benchmark suite are rather biased: they are either misaligned or aligned 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 15 gives the percentage of MDA instructions that are always misaligned, frequently misaligned and frequently aligned. As we can see in the figure, only about 4.5% MDA instructions are frequently aligned.

Besides, the multi-version method incurs overhead at runtime since the alignment checking instructions also consume cycles. As discussed in Section IV, generating multi-version code on basic-block granularity can help to decrease the runtime overhead.

C. *Overall Comparison*

Now that we have explored the design and parameter space for Direct Method, Static Profiling, Dynamic Profiling, Exception Handling, and DPEH, we can compare them against each other 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 16 (all results are normalized to the execution time of Exception Handling mechanism). As shown in the figure, the Exception Handling method is consistently better than other MDA handling mechanisms: 16% better than Dynamic Profiling, 10% better than Static Profiling and 68% better than Direct Method on average. However, the DPEH method gains additional 4.5% on top of the Exception Handling method.

As discussed in Section III, for applications that have frequent memory reference behavior changes, the Dynamic Profiling method may not work very well. As shown in

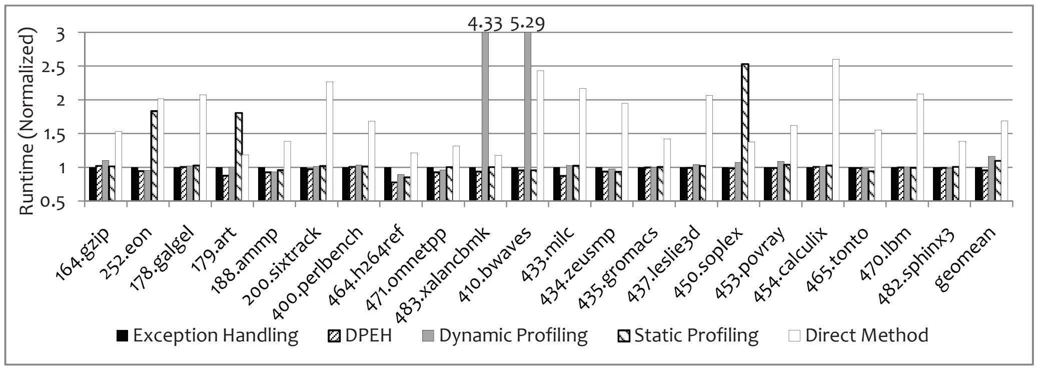


Figure 16: Performance for different MDA handling mechanisms

Table III: 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	433.milc	1.34E+08
252.eon	24630	434.zeusmp	1716
178.galgel	3436	435.gromacs	1820
179.art	3.12E+08	437.leslie3d	1716
188.ammp	0	450.soplex	9.33E+08
200.sixtrack	235950	453.povray	2.41E+08
400.perlbench	57874640	454.calculix	2609
464.h264ref	9347	465.tonto	116450
471.omnetpp	38979	470.lbm	0
483.xalancbmk	8.32E+09	482.sphinx3	1
410.bwaves	4.15E+10		

Table IV: The number of remaining MDAs while profiling with train input set

Benchmarks	Num of MDAs	Benchmarks	Num of MDAs
164.gzip	46	433.milc	6
252.eon	3.22E+09	434.zeusmp	644100
178.galgel	4930086	435.gromacs	0
179.art	3.6E+09	437.leslie3d	21168
188.ammp	0	450.soplex	4.03E+09
200.sixtrack	0	453.povray	0
400.perlbench	1244769	454.calculix	1.83E+08
464.h264ref	1020	465.tonto	262
471.omnetpp	48638638	470.lbm	0
483.xalancbmk	12761	482.sphinx3	0
410.bwaves	0		

Table III, when the heating 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 and 453.povray. As we can see in Figure 16, these six benchmarks are exactly the applications that suffer significant performance degradation compared with DPEH method (8% for 164.gzip, 14% for 179.art, 340% for 483.xalancbmk, 433% for 410.bwaves, 15% for 433.milc and 9% for 453.povray). For Static Profiling mechanism, the performance of most benchmarks is similar to DPEH method. Table IV summarizes the number of MDAs which are not detected while profiling with train input set. As we can see in Table IV, while running with ref input set, there are still a large number of MDAs in 252.eon, 179.art and 450.soplex. The performance of these benchmarks is significantly lower when compared with DPEH (91% for 252.eon, 13% for 179.art and 155% for 450.soplex). The Direct Method is generally worse than all others, simply because it indiscriminately increases the instruction overhead for all non-byte memory operations.

VII. 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 under-investigated issue is how to efficiently handle misaligned data accesses. Existing techniques either incur excessive instruction overhead, or cannot adapt to the change of memory access behavior at runtime.

In this paper, we have studied the strength and weakness of many existing MDA handling techniques. The direct translation method which translates every non-byte 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, but may fail to catch important MDA candidates during the profiling phase. Continuous profiling is more flexible in that it does not have to be limited to only the profiling phase. However, it may incur unacceptable high profiling overhead.

We have proposed an Exception Handler based mechanism which can be considered a light weight continuous profiling of MDA operations. As long as memory operations are executed without misaligned exceptions, they can run at full speed. When a misaligned exception occurs, the exception handler will translate this particular memory operation into the MDA code sequence. Based on this light weight continuous profiling approach, we further enhance it with dynamic profiling to catch many MDA candidates at the early stage. This combination significantly reduces the cost of MDA exception handling. We also make our method more adaptive by dynamically translate those memory operations that change between MDA and aligned accesses into a two version code. Furthermore, this method is improved by code repositioning and retranslation optimizations to maintain good instruction spatial locality. We have shown that this new method is more adaptive to program behavior change than all existing methods, as it outperforms the static profiling method by 14%, the dynamic profiling method by 20%, and the direct translation method by 73%, on 21 SPEC2000 and SPEC2006 benchmark programs with frequent misaligned data accesses.

ACKNOWLEDGMENT

This paper is supported by a project of the Nation Basic Research Program of china (No. 2005CB321602), a project of the National Natural Science Foundation of China (No. 60736012) and a project of the National High Technology Research and Development Program of China (No. 2007AA01Z110). This work is also supported by NSF CNS-0834599.

REFERENCES

[1] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, S. G. Robinson. *Binary translation*, Communications of the ACM, 36(2), 69–81, Feb. 1993.

[2] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. *FX!32: a Profile-Directed Binary Translator*, IEEE Micro, vol. 18(2),1998.

[3] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang and Y. Zemach. *IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium based systems*, Proceedings of the 36th International Symposium on Microarchitecture, 2003.

[4] F. Bellard. *QEMU, a Fast and Portable Dynamic Translator*, Proceedings of the 2005 USENIX Annual Technical Conference, 2005.

[5] V. Bala, E. Duesterwald, and S. Banerjia. *Dynamo: A transparent dynamic optimization system*, In PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, pages 1-12. New York, NY, USA, June 2000.

[6] W. Chen, S. Lerner, R. Chaiken, and D. Gillies. *Mojo: A dynamic optimization system*, In Proceedings of the ACM Workshop on Feedback-Directed and Dynamic Optimization FDDO-3, December 2000.

[7] V. Packirisamy, S. Wang, A. Zhai, W. Hsu, P. Yew. *Supporting Speculative Multithreading on Simultaneous Multithreaded Processors*, HiPC 2006, 148-158.

[8] N. Nethercote, J. Seward. *Valgrind: a framework for heavyweight dynamic binary instrumentation*, Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, June 10-13, 2007, San Diego, California, USA.

[9] P. P. Bungale, C. Luk. *PinOS: A Programmable Framework for Whole-System Dynamic Instrumentation*, Proceedings of the 3rd ACM/USENIX International Conference on Virtual Execution Environments(VEE 2007), June 2007.

[10] F. Qin, C. Wang, Z. Li, H. Kim, Y. Zhou, Y. Wu. *LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks*, MICRO 2006, 135-148.

[11] J. E. Smith, R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*, Elsevier, Morgan Kaufmann Publishers, May 2005, ISBN 1-55860-910-5.

[12] J. L. Hennessy, D. A.Patterson. *Computer Architecture: A Quantitative Approach*, 4rd ed. Elsevier, Morgan Kaufmann Publishers, SEP 2006, ISBN-13:978-0-12-370490-0, ISBN-10:0-12-370490-1.

[13] Dehnert, Grant, Banning, Johnson, Kistler, Klaiber, Mattson. *The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges*, Proceedings of the International Symposium on Code Generation and Optimization, 2003.

[14] B. Coon, G. D'Souza, P. Serris. *Patent: Pipeline Replay Support for Unaligned Memory Operations*, United States Patent, 7134001B1,2006.

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

[16] P. J.Drongowski, D. Hunter, M. Fayyazi, D. Kaeli. *Studying the Performance of the FX!32 Binary Translation System*, Proceedings of the 1st Workshop on Binary Translation , Newport Beach, CA, Oct, 1999.

[17] C. Zheng, C. Thompson. *PA-RISC to IA-64: transparent execution, no recompilation*, IEEE Computer, Mar, 2000.

[18] T. Feng, W. Chenggang, Z. Zhaoqing, Y. Hao. *Exception Handling in Application Level Binary Translation*, Journal of Computer Research and Development, 2006, 2166-2173.

[19] J. Li, Ch. Wu. *A New Replacement Algorithm on Content Associative Memory for Binary Translation System*, 1st Workshop on Architectural and Microarchitectural Support for Binary Translation, June, 2008.

[20] Compaq Computer Corporation. *Alpha Architecture Handbook*, 1998.

[21] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Vol. 1-3 2006.

[22] Standard Performance Evaluation Corporation. *SPEC CPU2000 Benchmarks*, http://www.specbench.org/osg/cpu2000.

[23] Standard Performance Evaluation Corporation. *SPEC CPU2006 Benchmarks*, http://www.specbench.org/osg/cpu2006.

[24] D. R. Ditzel. *Transmeta's Crusoe: Cool chips for mobile computing*. In IEEE, editor, Hot Chips 12: Stanford University, Stanford, California, August 13-15, 2000.