Leveraging Indirect Branch Locality in Dynamic Binary Translators

Balaji Dhanasekaran
University of Virginia
balaji.dhanasekaran@email.virginia.edu

Kim Hazelwood
University of Virginia
hazelwood@virginia.edu

Abstract

Dynamic Binary Translators (DBTs) have a wide range of applications like program instrumentation, dynamic optimization, and security. One of the main issues with DBTs is their performance overhead. A significant part of this overhead is caused by indirect branch (IB) translation. In this paper, we show that the percentage of instructions spent in translating indirect branches is as high as 50% of the total guest application's instructions. The time spent in indirect branch translation is so high that in some applications, bounding the code cache size actually results in an increased performance, since code cache flushes also remove stale indirect branch information along with the translated code. In order to improve the performance of indirect branches, we analyze the locality of indirect branch targets and show that the locality is as high as 70%. We propose an indirect branch translation algorithm which exploits this available locality. We analyze the performance of the proposed algorithm and show that the proposed algorithm achieves a hit rate of 73% compared to 46.5% with the default algorithm.

Keywords Dynamic Binary Translators, Code Cache, Indirect Branch Translation, Branch Target Locality

1. Introduction

Dynamic binary translators act as a middle layer between the guest application and the OS. They provide the ability to inspect, instrument, and translate instructions that are being executed. They allow the user to access the application's attributes that are available only at run time. For example, consider a compiler optimization like code motion, which may improve or hurt the performance of an application depending on how frequent a particular code path is executed. It is extremely difficult to make optimization decisions like these at compile time. But, with the help of binary translators, we can instrument the application at runtime and choose the appropriate optimization based on the execution characteristics of the application. DBTs provide enormous power in the hands of the users to analyze and optimize their applications at runtime. DBTs are also used in other applications like instrumentation (Pin [8] and DynamoRIO [2]), security (Strata [9]), dynamic translation (Rosetta [1]), and design space exploration (Daisy [4]).

One of the main problems with DBTs is their performance overhead. Figure 1 shows the relative performance of the SPEC2006 INT benchmarks when executed under the control of Pin. For some benchmarks like perlbench, the overhead of executing under Pin can be as high as 300%. One of the reasons for these high overheads is indirect branch translation. DBTs translate the guest code in units of basic blocks and traces and store them in the code cache. A basic block is a unit of instructions bounded by branch statements. A trace is a collection of basic blocks. As long as there are no branch statements, the control remains within the trace. When the application encounters a branch statement, the control is transferred back to the DBT. The DBT then translates the required instructions and transfers the control back to the application. In the case of direct branches, the context switch happens only during the first time the branch instruction is executed. When control is transferred to the DBT, it generates the target trace and patches the direct branch with the target trace's code cache address. In the case of indirect branches, this branch target can vary across executions. As a result, the branch target cannot be patched the way it is patched in direct branches. This causes a significant overhead in the DBT's performance.

The goal of this paper is to analyze the performance impact of indirect branches and propose an appropriate solution to mitigate this overhead. We show that there are performance trade-offs between indirect branch handling techniques like chaining and branch target hashing.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright © 2010 ACM [to be supplied]. . . $10.00

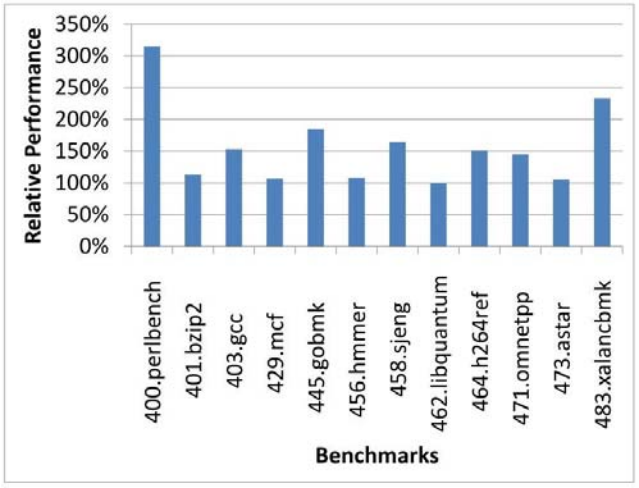

Figure 1. Normalized performance of the SPEC2006 INT Benchmark running under PIN. Performance is normalized with respect to native performance.

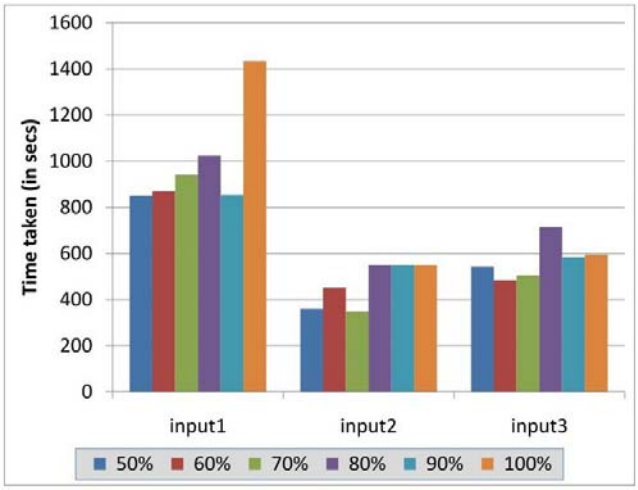

Figure 2. Performance of perlbench as the code cache size is varied.

Choosing the appropriate branch handling technique has a significant impact on performance. We also show that the indirect branch targets have a very high locality of about 73%. We propose an algorithm called the "Most Recently Used Branch Target" algorithm which exploits this locality by rearranging the indirect branch chain, such that the most recently used trace is checked first.

2. Motivation

One of the main motivations of this paper is an observation made by Hazelwood et al. [5] where they found that in some cases, restricting the code cache size actually improves the performance significantly. Figure 2 shows the performance of perlbench as we vary the code cache size. As shown in the figure, the best case performance is not obtained with an unbounded code cache. On the contrary, for some cases, restricting the code cache size actually increases the performance. Our analysis of this performance increase showed that the increase had nothing to do with the code cache size. Bounding the code cache increased the number of code cache flushes which also removed stale indirect branch information. And, this happened to be the cause of the performance increase. The other main insight in this paper is the high locality available in indirect branches. We show that there is a 70% probability that the target of an indirect branch during its current execution is the same as its target during the next execution.

3. Background

Pin is the dynamic binary translation system used in this paper. Pin is a multi platform DBT, mainly used for instrumentation purposes. Pin is available on multiple architectures including IA32 (32-bit x86), IA32e (64-bit x86), and Itanium. Pin is also available on multiple operating systems including Windows, Linux, and MacOS. The following subsections describe the code cache management and branch handling in Pin.

3.1 Runtime Translation and Code Caches in Pin

Pin translates the guest application in units of basic blocks and traces and maintains the translated code in a software code cache. Thereafter, execution happens only from the code cache and the original guest application is never executed directly. The code cache is bounded to 256 MB for 64-bit Pin and is unbounded for 32-bit Pin. For 32-bit Pin, the code cache management algorithm allocates more memory as and when new traces are generated. The drawback of this technique is that during the course of the application's execution, the code cache can get clogged with invalid traces and the valid traces can become scattered in the code cache. This defeats one of the important advantages of using a DBT, which is better instruction locality.

3.2 Branch Handling in Pin

Branch handling constitutes a significant portion of Pin's performance. A naïve way of translating branch instructions would be to transfer the control to the DBT whenever a branch statement is encountered. The DBT can then check whether the target trace is available in the code cache and generate it if the target trace is not present in the code cache. But, this approach of context switching between the application and the DBT whenever a branch statement is executed results in a significant performance drop. As a result, Pin uses techniques like trace linking and indirect branch chaining for translating branch instructions. We describe these approaches in the following subsections.

3.2.1 Direct Branch Translation

The first time a direct branch is executed, control is transferred back to Pin. Pin generates the required target trace and patches the branch instruction's target address with the target trace's code cache address. As a result, whenever the same branch instruction is executed again, control can be transferred directly from one trace to the other, without the intervention of Pin.

3.2.2 Indirect Branch Translation

In the case of indirect branches, the target address is present in a register or in a memory location. As a result, the target of an indirect branch can vary between different executions of the same branch instruction. Pin uses a combination of two techniques to translate indirect branches. First, each indirect branch instruction is associated with an indirect branch chain. When the indirect branch is first encountered, the control is transferred to the Pin VMM which generates the target trace. Pin also generates a compare and jump block and places it at the head of the generated trace. This compare and jump block compares the current indirect branch target with the target for which the trace was generated. If they match, control is transferred to the trace. If there is a mismatch, control is transferred to the next compare and jump block in the chain or to Pin. After the first target trace is generated, Pin patches the indirect branch stub to transfer the control directly transferred to the first compare and jump block in the chain.

The default number of traces present in the indirect branch chain is 16. When this chain length is exceeded, Pin generates a separate hash table for the indirect branch. This hash table is indexed using the branch target address and its default size is 256. Collisions in the hash table are resolved using chaining, i.e. each entry in the hash table has a separate chain of traces attached to it. Figure 3 illustrates the indirect branch translation mechanism used in Pin. Register *r1* holds the indirect branch's target address. The control is transferred to the compare and jump block of trace *t1* where *r1* is compared against *t1*. If there is a match, control is transferred to *c1*, the code cache address of trace *t1*. If there is a mismatch, then the control is transferred to the next compare and jump block. If there is no match in the indirect branch chain, then we index into the hash table using the branch target address and follow the chain associated with the indexed entry.

4. Experimental Setup

All experiments were conducted on 64-bit quad core dual Xeon 3.2 GHz systems with 8 GB of RAM running CentOS 4.8. All the systems have 32KB L1 I-cache and 4 MB L2 cache. We have analyzed the performance of 64-bit Pin 2.7-31931 using the SPEC2006 INT benchmarks with the reference inputs. We have also written our own pintools

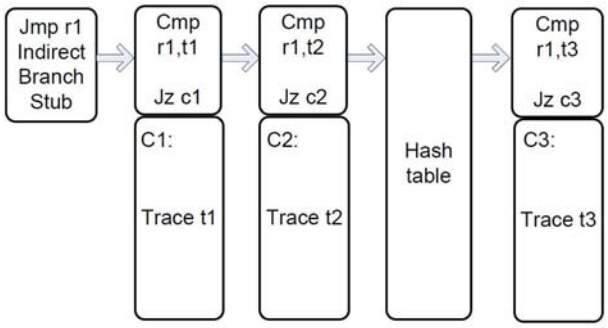

Figure 3. Indirect branch chain and the associated hash table. Register r1 holds the indirect branch's target address. t1, t2, and t3 are the trace's target addresses and c1, c2, and c3 are the corresponding code cache addresses.

to analyze indirect branches. Our pintool measures the number of trace traversals in the indirect branch chain and also the number of hash table accesses. We have modified 64-bit Pin 2.8-33543 to implement our MRU algorithm. All experiments were repeated for three iterations and the arithmetic mean of the three results have been reported.

5. Performance Impact of IB Translation

Our first step in analyzing indirect branches was to measure the performance impact of indirect branch translation. The direct approach would be to just measure the time spent in indirect branch translation. But, the number of instructions spent per indirect branch is very small. As a result, the time spent in measuring the indirect branch would be larger than the time spent in the indirect branch itself. So, we chose not to follow this approach since it is prone to high error margins.

Instead we analyze the number of instructions spent in indirect branch translation for the SPEC2006 INT benchmarks. We wrote our own pintool which instruments all the indirect branches in the guest application. We also added the appropriate code for simulating the overhead of indirect branch chains and target hash tables. We measured the number of hash table accesses and the number of traces traversed in the indirect branch chains. We also determined the number of instructions in the compare and jump block and also in the hash tables. Based on these values, we calculated the number of instructions spent in handling indirect branches and compared it with the total number of dynamic instructions in the guest application.

Listing 1 and Listing 2 represent the instructions used in the compare and jump block and in the hash table. The compare and jump block requires three instructions and the hash table requires eight instructions, five from Listing 2 and three from Listing 1 since we have to check whether the indexed entry matches the branch target.

Listing 1. Compare and jump block instructions

```
//move the trace's target to r13
mov r13, 0x3bbf30e887
//compare current target with trace's target
cmp r8, r13
//if not equal, then jump to the next
//compare and jump block
jnz 0x2a97c6f440
```

Listing 2. Hash table instructions

```
//move the branch target to rax
mov rax, r8
//shift rax
shr rax, 0x4
//move hash table index to rax
movzx rax, al
//move base to r13
mov r13, 0x2a95677800
//jump to base + index * sizeof(int)
jmp qword ptr [r13+rax*8]
```

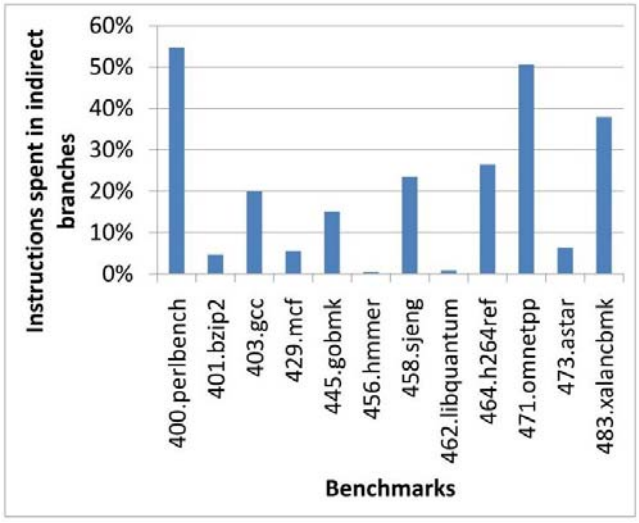

Figure 4. Instructions spent in resolving indirect branches as a percentage of total dynamic instructions of the guest application.

Figure 4 represents the number of instructions spent in resolving indirect branches as a percentage of total dynamic instructions of the guest application. For perlbench, the number of instructions spent on resolving indirect branches is more than 50% of the total dynamic instruction count. This means that if the performance of perlbench compared to native execution is 300%, then 25% of that overhead comes from resolving indirect branches.

Figure 5 and Figure 6 illustrate perlbench's performance and Pin's memory consumption as the chain length is varied. As we can see, as the chain length is reduced, the performance increases. The problem with the indirect branch chain of Pin is that it does not capture the notion of trace hotness (where a hot trace gets a lot of hits). The chain

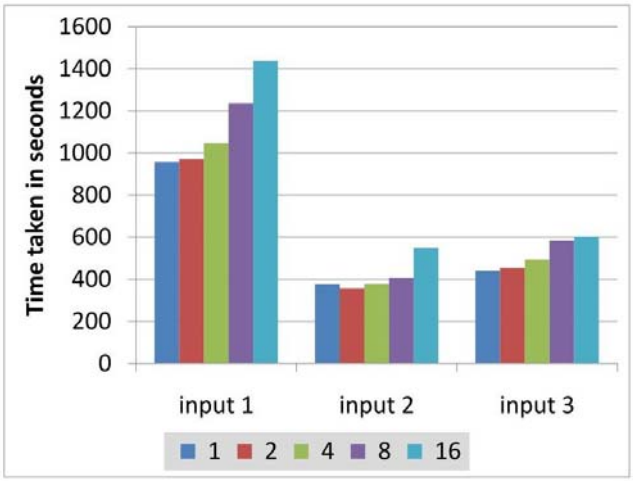

Figure 5. Performance of perlbench as we vary the chain length. (Longer chains perform worse.)

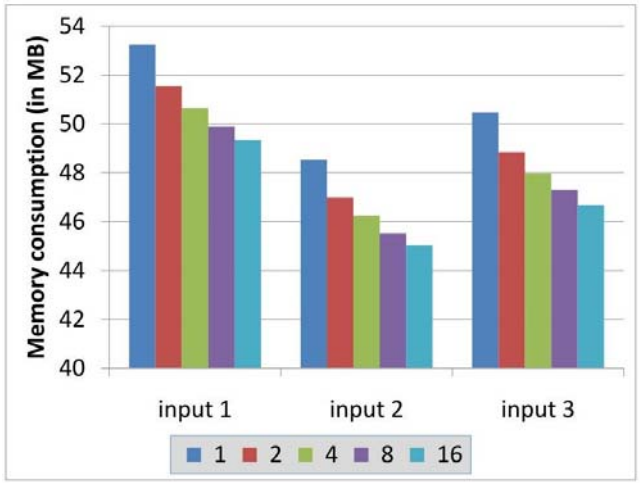

Figure 6. Memory consumption of Pin (in MB) for perlbench as we vary the chain length.

is never re-arranged in order to reflect the hit rates of the traces. The chain is built in FIFO order. As a result, if the hot trace gets generated last, then Pin will have to traverse multiple traces in the indirect branch chain before reaching the hot trace and this degrades performance. Reducing the chain length removes this unnecessary overhead. But, on the other hand, reducing the chain length also means that more indirect branches will have hash tables and hence an increased memory consumption. If we set the chain length to one, then all the indirect branches with more than one branch target will have a separate hash table and this will increase the memory consumption of Pin (as illustrated in Figure 6).

Figure 7 shows the relative performance of Pin with a chain length of 16 compared to chain length of one. We can see that the chain length of one achieves a best

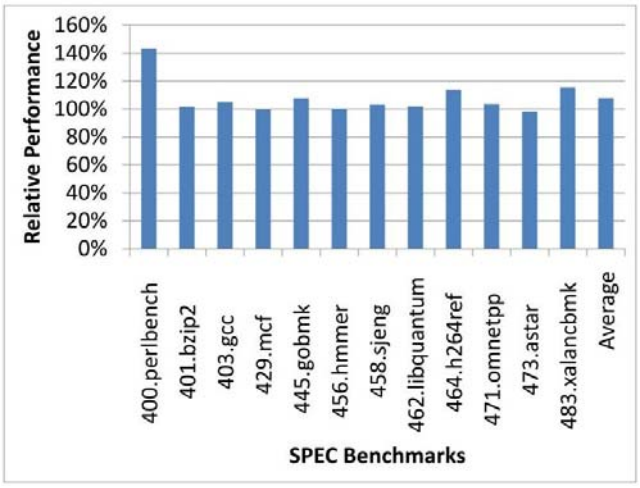

Figure 7. Relative performance of Pin with chain length of 16 compared to chain length of 1.

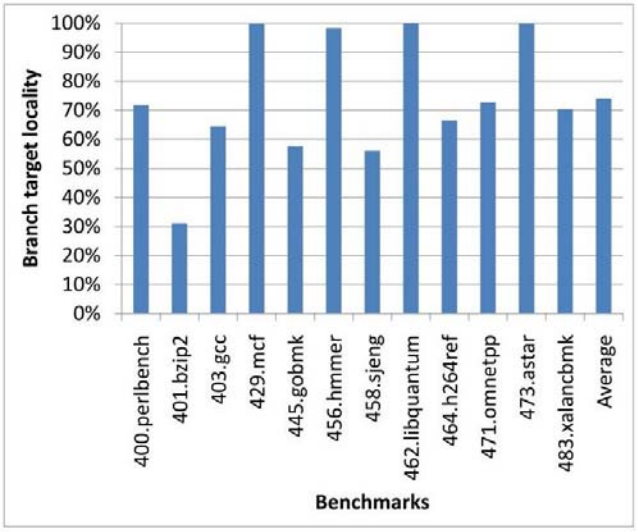

Figure 8. Indirect branch target locality in the SPEC2006 INT benchmarks. Percentage of indirect branch targets which are the same as the previous targets.

case speedup of 43.3% and an average speedup of about 7.7%. Except for two benchmarks, astar and mcf, all the other benchmarks show a speedup with chain length of one. The worst case slow down is only 2.7% compared to the best case speedup of 43.3%. These results show that there is a trade-off between traversing the indirect branch chain versus accessing the hash table directly. This trade-off can have a huge impact on performance and in the case of Pin, accessing the hash table directly gives a better performance than traversing the traces in the indirect branch chain.

6. Indirect Branch Target Locality

Instead of having a static chain where traces are inserted in FIFO order, we wanted to create a dynamic chain where the traces are rearranged based on their hotness. In order to do that, we analyzed the locality of indirect branch targets. For a given indirect branch instruction, we determined the percentage of branch targets that remain the same for two consecutive executions of the same branch instruction. If there is a high percentage of indirect branches where the current target is the same as the next target, then we can have a one-entry target cache per indirect branch that records the current branch target. The next time we execute this same indirect branch, we can check this target cache first before entering into the indirect branch chain.

Figure 8 illustrates the branch target locality in the SPEC2006 INT benchmarks. The average indirect branch locality in the SPEC benchmarks is 73.87%. This means that whenever an indirect branch is executed in any of the SPEC benchmarks, there is a 73.87% probability that the current branch target will be the same as the next branch target. This shows that the indirect branch targets have a very high locality. In the next section, we describe an algorithm which exploits this branch target locality.

7. Most Recently Used (MRU) Target Algorithm

For each indirect branch, the proposed algorithm allocates two words in memory: one for storing the current branch target and the other for storing the target's code cache address. Each time an indirect branch is executed, the algorithm first checks the single entry target cache. If there is a match, the control is transferred to the target code cache address directly. If there is a mismatch, then it is transferred to the indirect branch chain. The new algorithm also adds instructions at the beginning of every target trace to update the target cache. This algorithm is able to exploit the locality available in indirect branch targets.

Figure 9 illustrates the proposed MRU algorithm. At program startup, Pin replaces all indirect branches with a stub branching to itself. So, when an indirect branch is encountered, the control is transferred to Pin, which then allocates the memory required for the target cache and initializes it to zero. Pin also generates the code for comparing the branch target to the entry of the target cache. It then generates the compare and jump block. When the target trace is generated, Pin also adds instructions to update the target cache. The next time when the same indirect branch is executed, the control is transferred to the target cache code. If there is no match, then we move on to the first compare and jump block in the indirect branch chain. If there is a match, then the control is transferred to trace 't1' which updates the target cache. This makes

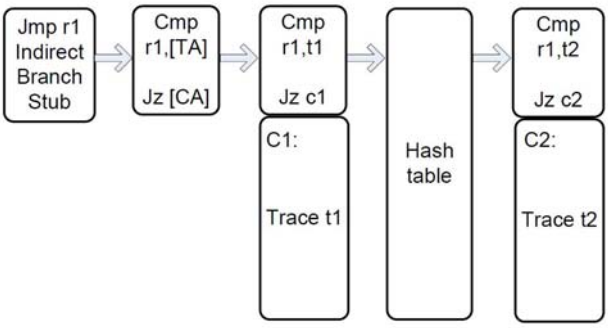

Figure 9. Indirect branch chain with the target cache. TA (Target Address) and CA (Code cache address) represent the memory locations containing the previous branch target and its corresponding code cache address.

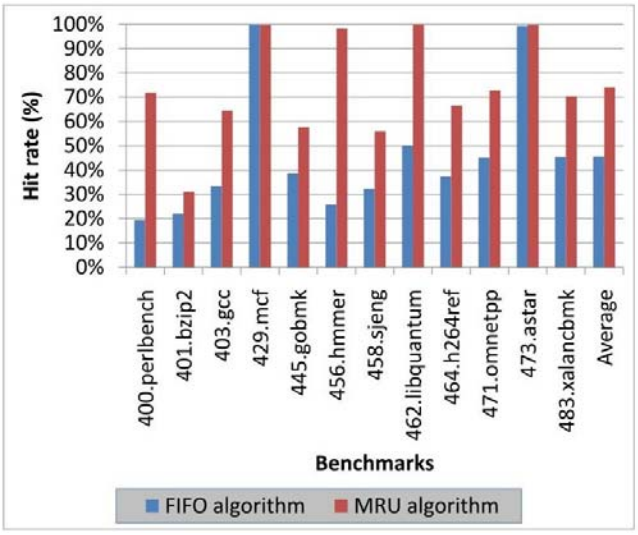

Figure 10. Comparisons of branch target hit rates.

sure that we have a hit in the target cache if we branch to the same target consecutively.

7.1 Simulation Results

We created a pintool to simulate the proposed MRU algorithm and determined the hit rates using the proposed algorithm. We compared this hit rate with the hit rate of the FIFO algorithm where the traces are inserted in FIFO order into the indirect branch chain.

Figure 10 illustrates the branch target hit rates in the various algorithms. In the case of MRU, the average hit rate in the target cache is about 73%. In the case of FIFO, the average hit rate in the first trace of the indirect branch chain is only 46.5%. Among the 12 SPEC benchmarks, the FIFO algorithm is able to match the hit rate of MRU only in astar and mcf. In all other benchmarks, MRU outperforms the FIFO algorithm. This shows that there is a huge potential for speedup using MRU compared to FIFO algorithm. Though MRU offers better hit rate, this doesn't guarantee a corresponding speedup. The final speedup depends upon several factors such as the net total number of instructions executed, cycles spent in memory stalls, etc. For example, in the compare and jump block of the indirect branch chain, all the addresses (target address and code cache address) are represented as immediate values in the instruction. But, in the case of the compare and jump block for MRU, all these addresses are present in the memory. So, one of the factors that might affect the speedup is the trade-off between waiting for memory (in the case of MRU) versus executing more instructions (in the case of FIFO algorithm). This is the reason why an increase in target hit rate need not necessarily mean a corresponding increase in speedup. In order to verify how the increased hit rate translates to increased performance, we developed a preliminary implementation of our algorithm in Pin.

7.2 Implementation Results

We have implemented our algorithm in the 64-bit version of Pin. Pin has a very well organized code base and provides a lot of APIs helpful for implementing new features. Pin has a lot of callbacks that get called when there is a context switch from the guest application to Pin. Pin also has one such call back for handling indirect branches. The first time the indirect branch is encountered, the stub transfers the control to Pin which then transfers it to the indirect branch callback. The indirect branch callback checks whether the chain length for the current indirect branch is less than the maximum chain length and then generates the compare and jump block and the corresponding target trace. If the maximum chain length is exceeded, the indirect branch callback generates the code for the hash table. We modified the indirect branch callback to add a compare and jump block for the MRU cache when the chain length is zero. Thereafter, the usual indirect branch chain code gets executed. We also had to make changes in a few other places for passing the target cache address to the indirect branch callback.

Figure 11 and 12 illustrate the relative performance of an implementation of MRU compared to FIFO. Figure 12 shows the relative performance for a chain length of 16 and Figure 11 for a chain length of two. For the shorter chain length, the performance of MRU is approximately the same as the performance of FIFO. For the longer one, MRU shows a best case speedup of 3.4%. But, on an average, the performance of MRU is less compared to FIFO.

When we analyzed the performance of MRU, we noticed that though MRU resulted in an increased hit rate, the total dynamic instruction count of MRU is higher compared to FIFO. This is the reason for why MRU's performance did not improve significantly compared to FIFO. One of our future works is to improve the implementation of MRU.

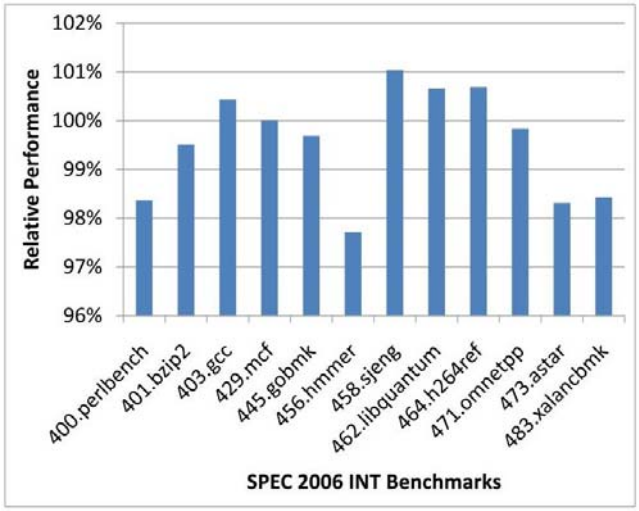

Figure 11. Relative Performance of MRU algorithm with respect to FIFO with chain length of 2.

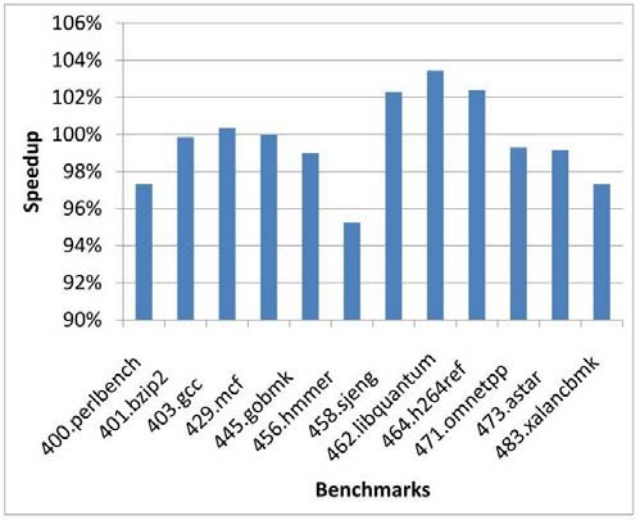

Figure 12. Relative Performance of MRU algorithm with respect to FIFO with chain length of 16.

8. Related Works

A lot of work has been done in dynamic binary translation and its subfields like code cache management, register allocation, indirect branch translation etc. One of the recent extensive analyses of indirect branches is work done by Hiser et al. [6]. They analyzed various indirect branch translation algorithms and their performance impact in Strata. Their work also mentions an approach similar to MRU algorithm. But, in their work, instead of having a separate memory location for holding the target address and code cache address, they directly patch the instructions that perform the target cache comparison. This causes the instruction cache to become incoherent and results in a huge performance drop. Moreover, their work does not discuss indirect branch target locality.

Smith and Nair [10] provide an overview of the various indirect branch translation techniques used in Dynamic Binary Translators. Bruening et al. [3] analyzed the implementation of indirect branches in DynamoRIO. In terms of hardware support, Kim et al. [7] analyzed the architectural support required for indirect branch translation. One of their proposals is a hardware lookup table for branch targets, which also exploits the locality available in indirect branches.

9. Conclusion & Future Work

In this study, we analyzed the performance impact of indirect branches, and demonstrated that the dynamic instructions spent in indirect branches can be as high as 50% of the total dynamic instructions of the guest application. We have shown that there are trade offs between indirect branch handling techniques and choosing the appropriate technique can have a huge impact on performance. We have also shown that the indirect branches have a very high locality of about 73%. We have proposed an algorithm for exploiting this locality and have shown that for longer indirect branch chains, the proposed algorithm achieves a best case speedup of 3.4%. Our main future work is to improve the implementation of MRU to reduce the dynamic instruction count. We would also like to implement MRU in other DBTs like DynamoRIO and Strata and determine the performance impact in those DBTs.

References

[1] Apple Inc. 2009. Universal binary programming guidelines, second edition. 57–66.

[2] Bruening, D., Duesterwald, E., and Amarasinghe, S. 2001. Design and implementation of a dynamic optimization framework for windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*. Austin, TX.

[3] Bruening, D., Garnett, T., and Amarasinghe, S. 2003. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, San Francisco, CA, USA, 265–275.

[4] Ebcioglu, K. and Altman, E. R. 1997. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture*. Denver, Colorado, United States, 26–37.

[5] Hazelwood, K., Lueck, G., and Cohn, R. 2009. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *International Symposium on Memory Management (ISMM)*. Dublin, Ireland, 20–29.

[6] Hiser, J. D., Williams, D., Mars, J., Childers, B. R., Hu, W., and Davidson, J. W. 2007. Evaluating indirect branch handling mechanisms in software dynamic translation

systems. In *Intl. Symp. on Code Generation and Optimization*. San Jose, California, 61–73.

[7] KIM, H.-S. AND SMITH, J. E. 2003. Hardware support for control transfers in code caches. In *36th annual IEEE/ACM International Symposium on Microarchitecture*. San Diego, California, 253–264.

[8] LUK, C., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V., AND HAZELWOOD, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Chicago, IL, USA, 190–200.

[9] SCOTT, K., KUMAR, N., VELUSAMY, S., CHILDERS, B., DAVIDSON, J., AND SOFFA, M. 2003. Retargetable and reconfigurable software dynamic translation. In *International Symposium on Code Generation and Optimization*. San Francisco, CA, 36–47.

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