TR-IIS-12-003

Improving Region Selection Through Early-Exit Detection

Chun-Chen Hsu, Pangfeng Liu, Jan-Jan Wu,
Chien-Min Wang, Ding-Yong Hong, Wei Chung Hsu

Apr. 24, 2012 || Technical Report No. TR-IIS-12-003
http://www.iis.sinica.edu.tw/page/library/TechReport/tr2012/tr12.html

Improving Region Selection Through Early-Exit Detection

Chun-Chen Hsu, Pangfeng Liu
Department of Computer Science and Information Engineering, National Taiwan University
{d95006,pangfeng}@csie.ntu.edu.tw

Jan-Jan Wu, Chien-Min Wang, Ding-Yong Hong
Institute of Information Science, Academia Sinica
{wuj,cmwang,dyhong}@iis.sinica.edu.tw

Wei Chung Hsu
Department of Computer Science, National Chaio Tung University
hsu@cs.nctu.edu.tw

ABSTRACT

Many dynamic binary translation (DBT) systems and just-in-time compilers target traces, i.e. frequently-taken execution paths, as code *regions* to be translated/optimized. The Next-Tail-Execution (NET) trace selection method used in HP Dynamo is an early example of such techniques. Many current trace optimization schemes are actually variations of NET. These NET-like trace optimizations work very well for most traces, but they also suffer the same problem: the selected traces may contain a large number of early exits that could branch out in the middle of traces. If early exits are taken frequently during program execution, the benefit of trace optimization could be lost due to the overhead of costly compensation code in the trace epilogue. We refer to traces/regions with frequently taken early-exits as *delinquent traces/regions*. Our empirical study shows that at least 9 of the 12 SPEC CPU2006 integer benchmarks have delinquent traces, i.e., if we use NET to select traces, each of these nine benchmarks will take more than 100 early exits per million executed instructions in their traces.

In this paper, we significantly improve the performance of NET by merging delinquent traces into larger code regions. We propose a light-weight region formation technique called *Early-Exit Guided region selection (EEG)*to improve the performance by iteratively detecting and merging delinquent regions into larger code regions. Hardware assisted dynamic profiling is first used to identify hot code regions without incurring significant runtime overhead. Key software counters are then instrumented at the exit points of the hot regions to detect early exits. When a counter exceeds certain threshold value, the code region that begins at the branch target of that early exit is merged into the main code region. We also employ a heuristic to decide whether it is beneficial to merge the selected regions or not. We will not merge two regions if the cost of spill code is too high for the merged code.

We implement our EEG algorithm in two LLVM-based parallel dynamic binary translators. These two parallel dynamic binary translators are for ARM and IA32 instruction set architecture (ISA) respectively, and both use multiple compilation threads to compile different code regions concurrently. We evaluate the performance of EEG with two benchmark suites: the SPEC CPU2006 single-threaded benchmark suite with *reference inputs*, and the PARSEC multi-threaded benchmarks with *native inputs*. The experimental results show that, compared to NET, EEG can achieve a performance improvement of up to 67% (13% on average) for SPEC CPU2006 integer benchmarks, and up to 20% (10% on average) for PARSEC multi-threaded benchmarks.

1. INTRODUCTION

Dynamic binary translation (DBT) is a just-in-time (JIT) compilation from binary code of a *guest* ISA to a *host* ISA. Cross-ISA binary translators enable an application to migrate from one hardware platform to another, or can provide a virtualized platform to run an application without the specific hardware. For example, DEC FX!32 enables an application to migrate from IA-32 to Alpha, and Intel IA-32EL [4] enables an application to migrate from IA-32 to Itanium. Other migration examples include Apple Rosetta. QEMU [5], VMWare use binary translation technique to provide server virtualization.

DBT needs to generate costly recovery code in the epilogue to handle the case when the uncommon conditional path is taken, referred to as an *early exit* from the trace. If early exits are taken frequently, the benefit of trace optimization could be lost due to the overhead of costly compensation code in the epilogue of the trace. We refer to traces with frequently taken early-exits as *delinquent traces*.

Many DBT systems [7, 10, 6] follow the well-known runtime trace selection algorithm, called Next-Tail-Execution (NET), developed for HP Dynamo [3]. Instead of profiling execution path at runtime, NET forms a trace by selecting the blocks that are most recently executed. The idea is that when an instruction becomes hot, it is likely that the following instructions are also hot.

However, since NET does not use any edge profiling information to select traces adaptively, early exits may occur when program behavior changes. For example, in our empirical study, we found that NET builds a trace with many frequently taken early exits for the SPEC CPU2006 456.hmmer benchmark. In 456.hmmer, there is a frequently executed for-loop that contains numerous compare-and-jump instructions (see Figure 1(b)).

NET separates the for-loop in Figure 1(b) into four traces as shown in Figure 1(a). Each rectangle represents a trace. The percentage of total execution time of each trace is shown on the left top corner of the trace. The probability of how likely an exit will be taken is also shown.

As shown in Figure 1(a), NET builds a trace for a loop starting at 0x80522be, but the probability of taking an early exit during the loop execution is 98%. Such a high probability for an early exist certainly eliminates the performance benefit that was expected from the loop trace. Our proposed region formation technique (to be described later) will merge these four traces into a large region shown in Figure 1(b). This region formation scheme removes early exits and forms a region that accounts for 81% of the total execution time

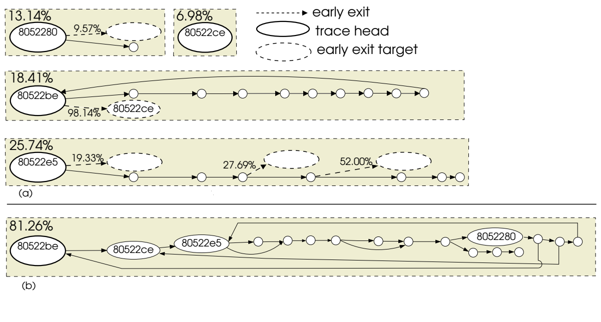

Figure 1: An example of delinquent traces of NET in 456.hmmer benchmark.

The insight we gain from this example is that when the program execution takes early exits frequently, the program behavior has changed, and the hot paths might have changed as well. By merging a region with another region that begins at the branch target of an early exit, we could form a larger region with less early exits. By merging delinquent traces/regions into larger regions, we can improve the performance.

In this paper, we propose a light-weight region formation technique called Early-Exit Guided (EEG) region selection to detect and merge delinquent regions. There are two key issues in our region formation technique: (1) which region should be merged, (2) when to merge a region. The simplest approach to address the first issue is to instrument counters into all traces and regions. However, this approach is prohibitively expensive and may merge too many regions that are not frequently executed. Instead, we employ hardware-assisted dynamic profiling to select hot regions and to avoid monitoring and merging unimportant regions. To address the second issue, we monitor regions by instrumenting counters to detect early exits. When the counter of an early exit exceeds a threshold, we merge this region with a region that begins at the branch target of the early exit. We also employ a heuristic to decide whether it is beneficial to merge selected regions. We will not merge regions if there is too much spill code in the translated code.

The main contributions of this work are as follows:

1. This work is the first effort on empirical study of the early exit problem in NET. Our profiling and experiment results show that NET generates substantial amount of delinquent traces, and that more than 100 early exits are taken for every million executed instructions in 9 of the 12 SPEC CPU2006 integer benchmarks.

2. We propose an elegant and effective region selection technique to solve the delinquent trace problem in NET. The proposed Early-Exit Guided region selection (EEG) uses hardware-assisted dynamic profiling and instrumented software counters to detect and merge delinquent traces/regions into larger regions.

3. We implement the region selection strategy in two LLVM-based parallel dynamic binary translators. These two parallel dynamic binary translators are for ARM and IA32 instruction set architecture (ISA) respectively, and both use multiple compilation threads to compile different code regions concurrently. By off-loading

region compilation to different cores, our system can perform more aggressive and sophisticated optimizations at the region and trace-level with very little overhead to DBT.

4. Experimental results show that the performance of the code produced by EEG is better than the code produced by NET, by up to 67% (13% on average) for SPEC CPU2006 integer benchmarks, and achieves up to 20% (10% on average) improvement for PARSEC multi-threaded benchmarks. For ARM-to-x86_64 DBT, it has only 2.06X slowdown compared to native execution of SPEC CPU2006 integer benchmarks, as opposed to 2.35X slowdown using the NET scheme. For IA32-to-x86_64, it has 1.8X, 2.12X, and 1.96X slowdown with SPEC CPU2006 integer, floating, and PARSEC multi-threaded benchmarks respectively, compared to 2.0X,2.15X, 2.1X slowdown using the NET scheme.

The rest of the paper is organized as follows. Section 2 presents our Region-Based Multi-threaded dynamic binary translator. Section 3 describes our early exit detection technique and early-exit guided region selection strategy. Section 4 presents our experimental results. Section 5 describes related work, and Section 6 gives some concluding remarks.

2. REGION-BASED MULTI-THREADED DYNAMIC BINARY TRANSLATOR

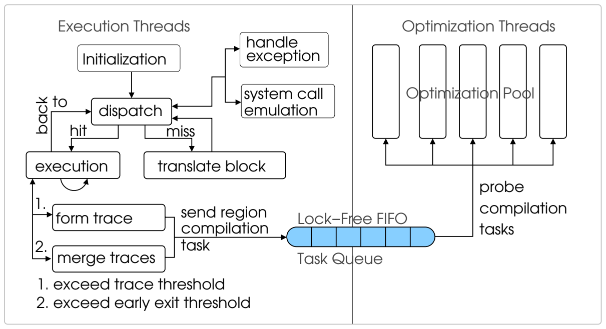

Figure 2: Control flow of execution threads and optimization threads

In this section, we describe the design of our region-based multi-threaded dynamic binary translator. We use the *LnQ* [12] dynamic binary translation framework to build our region-based multi-threaded dynamic binary translation system. LnQ uses LLVM [1] compilation infrastructure to build the backend just-in-time compilers, and uses QEMU [2] as the emulation engine. We inherit the retargetability of LnQ, and extend the framework to accommodate a optimization thread pool. We implement the early-exit guided region selection on top of this framework. Figure 2 shows the control flow of our region-based multi-threaded dynamic binary translators.

We use blocks, traces, and regions to refer to blocks, traces and regions of the *guest program*. We use *fragment* to denote a code segment translated by our binary translator. Therefore we have *block fragment*, *trace fragment*, and *region fragment*, each contains translated code of a block, a trace, or a region respectively.

Each fragment has prologues to load the guest architecture states, such as guest CPU registers, to host registers before execution. Also, we have epilogues to store modified dirty states into memory before leaving fragments. To achieve retargetability, prologues and epilogues are necessary because the host CPU architecture may not has enough registers to hold all registers of the guest CPU. Therefore, we do not specify register binding between guest architecture states and host architecture states. Each fragment has its own register binding scheme decided by the LLVM register allocator.

Our dynamic binary translation system has execution threads and optimization threads. Execution threads are responsible for running the translated segments and translating block segments. That is, if an execution thread reaches a new guest basic block during execution, the execution thread generates block fragments using LLVM compiler. Optimization threads generate traces and regions fragments with LLVM JIT compilation. All threads have their own LLVM compiler to compile blocks, traces or regions, and threads can compile blocks, traces or regions concurrently. To enable concurrent DTB compilation, each thread has its own memory chunk to store the translated fragments. Although we use multiple memory chunks for translated fragments, we adopt shared code cache design in our DBT system. Such design allows the execution threads to transfer execution among all chunks. Thus, logically we have one software code cache for all translated fragments.

Furthermore, our DBT system is able to separate trace compilations from program execution. By running multiple optimization threads to compile traces or regions concurrently, we off-load the compilation to other CPU cores and, hence, the execution threads are not interrupted. Execution threads may create region compilation tasks and send them to the *Task Queue* when traces or regions are formed as described in Section 3. We use a lock-free concurrent FIFO queue [13] to implement our task queue so that execution threads can insert trace/region compilation tasks into the queue while the optimization threads can probe those tasks from the queue without locks. This design enable our DBT to preserve good scalability when it runs multi-threaded guest applications.

3. EARLY EXIT INDEX AND EARLY-EXIT GUIDED REGION SELECTION

In this section, we first describe the NET algorithm used in our system, and how we define an *early exit index* to quantify how often early exits are taken in a trace. Finally we describe our early exit guided region selection technique.

3.1 Trace Selection Algorithm

We adopt a modified NET algorithm similar to [6] to builds traces. The difference to the NET algorithm is that our algorithm considers all basic blocks as possible trace heads, while the NET algorithm only considers potential loop beginning blocks as trace heads. Our algorithm has two advantages. First, the NET algorithm [3] was designed for single core. Because of the limited capacity of a single core, NET can only selectively build traces. In contrast, our algorithm can take advantage of multi-core platforms and use multiple optimization threads to compile different traces concurrently. Therefore, it can afford to try all blocks as trace heads. Second, applications such as chess programs may contain hot regions that are not loops. By considering all blocks as possible trace heads, we can discover more hot traces than NET does.

We use counters to record the number of times each block is executed. A block becomes a trace head when the block has been executed more than a pre-determined number of times. We form a trace by appending blocks along the execution path until one of the following terminal conditions is met.

1. A branch to the trace head is taken.
2. The number of blocks exceeds a threshold.
3. The next block is the head of another trace.
4. A guest system call instruction is encountered.

3.2 Early Exit Index

We first define an early exit of a code region. A code region can be represented by a control flow graph where a node represents a basic block and an edge from node A to node B indicates that the execution can proceed from block A to block B. A trace can be a simple path or a cycle in the control flow graph that has single entry and multiple exits. If a trace is a path, then all exit edges along the path are early exits except the exit edges of the last node of the trace. If a trace is a cycle, all exit edges are early exits except the edges of the last node that leaves the trace.

We define an Early Exit Index (EEI) to measure the frequency of an early exit being taken in a trace. Formally, EEI is the number of early exits being taken for every million instructions executed.

$$EEI = \frac{\sum_{i \in \Gamma} n_i \times \rho_i}{N}$$

Let Γ be the set of traces, n_i be the number of times early exits being taken in trace i. Let ρ_i be the percentage of instructions executed in trace i, and N be the number of million instructions executed. We use the number of early exits in trace i, ρ_i, as the weight so that the index indicates the average number of early exits being taken per million instructions executed in all traces.

3.3 Early-Exit Guided Region Selection

In this section, we describe our proposed Early-Exit Guided (EEG) region selection scheme. It detects and merges regions that have frequently taken early exits. The key issues in EEG are (1) how to efficiently detect delinquent regions; and (2) when to merge them at runtime. We address them as follows.

The simplest approach to address the first issue is to instrument counters into all traces and regions. However, this approach is inefficient and may merge too many regions that are not frequently executed. Instead, we use a dynamic profiling approach with the help of on-chip hardware performance monitor (HPM) to select hot regions.

We create a profiling thread called *profiler* at the beginning of execution to perform dynamic profiling. The profiler collects program counters per one million retired instructions. When a threshold number of samples are collected, the profiler accumulates the sample counts for each trace to determine the degree of *hotness* of each trace. The hotness of a trace is measured by the following equation.

$$H_T = \max\{\alpha, \beta\}$$

Here, α is the percentage of instructions executed in the trace during the last sampling period, and β is the percentage of instructions executed in the trace during the entire execution. Intuitively, α represents the hotness of the trace during in the last period, and β represents the accumulated hotness during the entire execution. We choose the maximum of α and β as its hotness measure.

When the hotness of a trace exceeds a threshold, we start monitoring this trace by instrumenting counters to its early exits. Currently, we only instrument early exits of conditional branches. If a counter exceeds a predefined threshold, it indicates some early exit starts being taken frequently. The monitored region is merged with the region that begins at the branch target of the early exit. The merged region is then translated by our LLVM-based DBT, and then the monitored region is replaced by the merged region. Note that the newly merged region is not monitored until it becomes hot again.

We argue that the overhead of the instrumentation is negligible in that early exits should be rarely taken. A frequently taken early exit would have triggered region formation when the counter exceeded a threshold, so the overhead of instrumented code is negligible.

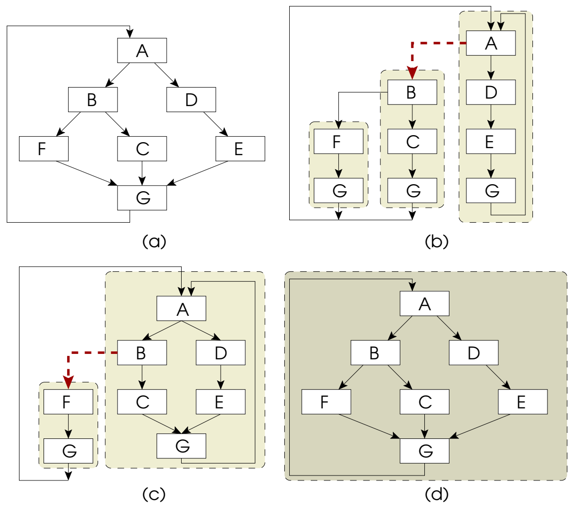

Figure 3: Illustration of region selection. Figure 3(a) is the CFG of a hot region in a guest application. Figure 3(b) shows three traces formed by NET.

We use Figure 3 as an example to illustrate our region selection strategy. Figure 3(a) is the control flow graph (CFG) of a hot region in a guest application. During execution, it first forms three traces as in Figure 3(b). Trace A would be the first selected for early exit detection since a loop is likely to become hot. Thus the early exit of Trace A, plotted as the red dashed line from Trace A to Trace B, is monitored with an instrumented counter.

We merge Trace A and Trace B to form a region when the early exit is taken frequently. As shown in Figure 3(c), a region consisting of traces A and B is formed. After the segment of Region A is generated, we replace Trace A with Region A so that Trace F now branches to Region A rather than to Trace A. Figure 3(d) shows the final result after detecting early exit of B to F.

	SPEC CPU2006	PARSEC
CPU	Intel Core2	Intel X5550
GHz	3.33	Intel 2.67
#Chips	1	2
#Cores	4	4
#Threads	2	2
L2 Cache	128KB	128KB
L3 Cache	8192KB	8192KB
Memory	12GB	20GB
GCC Version	4.3.4(IA32)/4.4.2(ARM)	4.3.4(IA32)
GCC Flags	-m32 -O2(IA32) -O2(ARM)	-m32 -msse3 -ftree-vectorize
Input Size	Ref. Inputs	Native Inputs
OS	Linux Gentoo 64 bit kernel 2.6.30	

Table 1: Summary of experiment environments

3.4 Spill Index of a Region

The benefits of EEG region selection come from eliminating the overhead caused by frequently-taken early exits and potential optimization opportunities from a larger region. Despite the fact that we can always eliminate the overhead of frequently-taken early exits via regions merging, we may not always have potential optimization opportunities from the merged region. In particular, when the quality of the translated code of a region is not good enough, it is not beneficial to merge such region.

The *Spill Index* is the percentage of spill code (i.e., the code for load/store operations between registers and stack) in the translated code. In our DBT system, we use Spill Index to assess the quality of the code generated by the LLVM compiler, and to decide whether the merging process for a region should be terminated. The definition of Spill Index is reasonable because high percentage of spill code often forestalls good performance due to improper register allocation of LLVM compiler. A region with high percentage of spill code is not suitable to be merged into a larger region.

4. EXPERIMENTS

In this section, we evaluate the performance of EEG in our LLVM-based parallel DBT systems. We first describe our measurement methodology, and then present detailed analysis of overall performance, the early exit index and its impact on performance, and the scalability of our DBT system.

We evaluate the performance of EEG with two benchmark suites: the SPEC CPU2006 single-threaded benchmark suite with reference inputs, and the PARSEC multi-threaded benchmarks with native inputs. We run the SPEC CPU2006 benchmarks on an Intel Core2 CPU 975 machine. Table 1 gives the hardware specification of the experiment platforms. We compile benchmarks into IA32 and ARM instructions, and run the binary code on our dynamic binary translators, IA32-to-x86_64 and ARM-to - x86_64. For ARM-to-x86_64, we run SPEC CINT2006 integer benchmarks because SPEC 2006 floating-point benchmarks are not supported on ARM platforms yet. We do not report the results of `h264ref` because the SPEC runspec tool reports a mis-match error even when it runs `h264ref` in a native ARM machine.

For SPEC CFP2006, we collect experimental results of the 8 widely used benchmarks, `povray, GemsFDTD, lesliesd, lbm, calculix, cactusADM, soplex, and dealll`. For PARSEC benchmarks, we run the IA32 version and collect the results of 8 benchmarks. We report the median of 5 runs for all performance metrics.

We run *Trace-* and *Region-* configuration DBT's in our experiments and use Trace-configuration DBT as our baseline. Trace-configuration DBT uses NET to select traces as described in Section 3. We set block count threshold to 50 and allow at most 16 blocks in a trace. Region-configuration DBT uses NET and EEG region selection algorithm as described in Section 3.3. We use Perfmon2 [14] for hardware-assisted dynamic profiling. The early exit threshold is set to 1000 and the Spill Index is set to 15%, i.e. when the percentage of spill code in the translated code exceeds 15%, the region can not be further merged. For Trace- and Region-configurations DBT's, we use two optimization threads to compile regions.

4.1 Performance Results of SPEC CPU2006

The performance results are shown in Figure 4. The average performance improvement of Region over Trace are 13.1%, 13.7% and 1.5% for SPEC CINT2006 IA32 binary, SPEC CINT2006 ARM binary, and SPEC CFP2006 IA32 binary, respectively. For SPEC CINT2006 - IA32, Region achieves up to 67% and 11.9% performance improvement over Trace for integer and floating benchmarks, respectively. For ARM-to-x86_64, Region improves the performance of the sjeng benchmark by 49%. Among all CINT2006 benchmarks, only xalancbmk and gcc have performance slowdown. Among all SPEC CFP2006 benchmarks, only povray has performance slowdown. These performance penalties rang from 0.60%. to

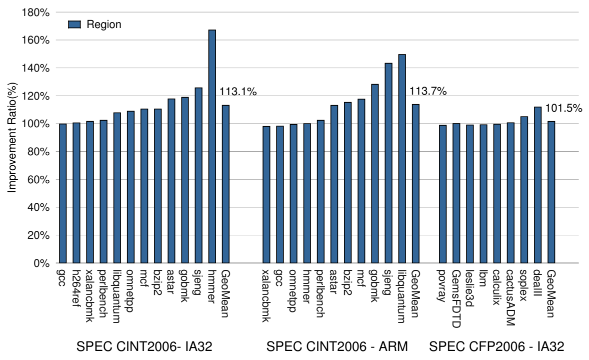

Figure 4: Performance of EEG region selection with NET.

2.1%.

Recall that the benefits of EEG region selection come from eliminating the overhead caused by frequently-taken early exits and potential optimization opportunities from larger regions. Therefore, to further understand the performance gain of our region selection, we use hardware performance monitor to profile the execution on Trace- and Region-configuration DBT's. We notice that one major contribution to performance gain comes from the reduction of memory instructions, as shown in Table 2.

4.2 Early Exit Index

We then measure the Early Exit Indices of benchmarks in Trace configuration. We insert counter for each side exit of traces to collect the number of early exits taken of each trace. We also use Perfmon2 [14] to measure the percentage of execution of traces by sampling program counters per one million executed instructions. The EEI results are shown in Figure 5, where EEI_C and EEI_I are early exit indices for conditional and indirect branches, respectively. First, we observe that 9 benchmarks in SPEC CINT2006 have EEI larger than 100 in both ARM and IA32 benchmarks.

Most of these benchmarks benefit from our EEG region selection because EEG eliminates the overhead caused by frequently-taken early exits. For example, IA32 `hmmer` benchmark and ARM `libquantum` have high EEI values and both achieves 67% and 49% performance improvements. As shown in Table 2, both applications have large percentages of memory operations reduced via merging delinquent regions. For IA32 `hmmer`, 36% of stores and 75% of loads are reduced; and for ARM `libquantum`, we also observe 59% and 73% of reduction in stores and loads.

As mentioned in Section 3.4, the benefits of EEG

CINT2006	IA32-to-x86_64			ARM-to-x86_64			CFP2006	IA32-to-x86_64		
Benchmarks	Reduced Mem. Ops		Improved	Reduced Mem. Ops		Improved	Benchmarks	Reduced Mem. Ops		Improved
	Stores	Loads	Perf.	Stores	Loads	Perf.		Stores	Loads	Perf.
perlbench	1.60%	13.46%	2.41%	-19.23	-6.55%	2.40%	cactusADM	-0.08%	0.06 %	0.59%
bzip2	7.64%	33.04%	10.50%	21.59%	36.99%	15.22%	leslie3d	-0.86%	-0.31%	-0.97%
gcc	0.98%	5.50 %	-0.25%	-3.90%	-2.57%	-1.73%	dealII	9.26%	19.56%	11.90%
mcf	23.62%	50.93%	10.46%	31.17%	67.07%	17.58%	soplex	12.24%	18.49%	4.95%
gobmk	10.66%	26.90%	18.84%	0.16%	10.93%	28.19%	povray	-2.56%	3.75 %	-1.19%
hmmer	36.21%	75.70%	67.14%	0.32%	3.75 %	-0.07%	calculix	-3.08%	-1.54%	-0.40%
sjeng	25.55%	42.09%	25.64%	15.51%	28.25%	43.29%	GemsFDTD	-0.14%	0.05 %	-1.02%
libquantum	19.67%	57.66%	7.77%	59.36%	73.42%	49.52%	lbm	-0.81%	-0.53%	-0.86%
h264ref	-7.83%	13.42%	0.49%	N/A	N/A	N/A				
omnetpp	16.00%	21.39%	8.91%	-4.18%	0.48 %	-0.69%				
astar	13.78%	45.62%	17.70%	13.89%	31.69%	13.06%				
xalancbmk	3.52%	5.30 %	1.60%	2.29%	14.18%	-2.08%				

Table 2: Reduced memory operations by region formation

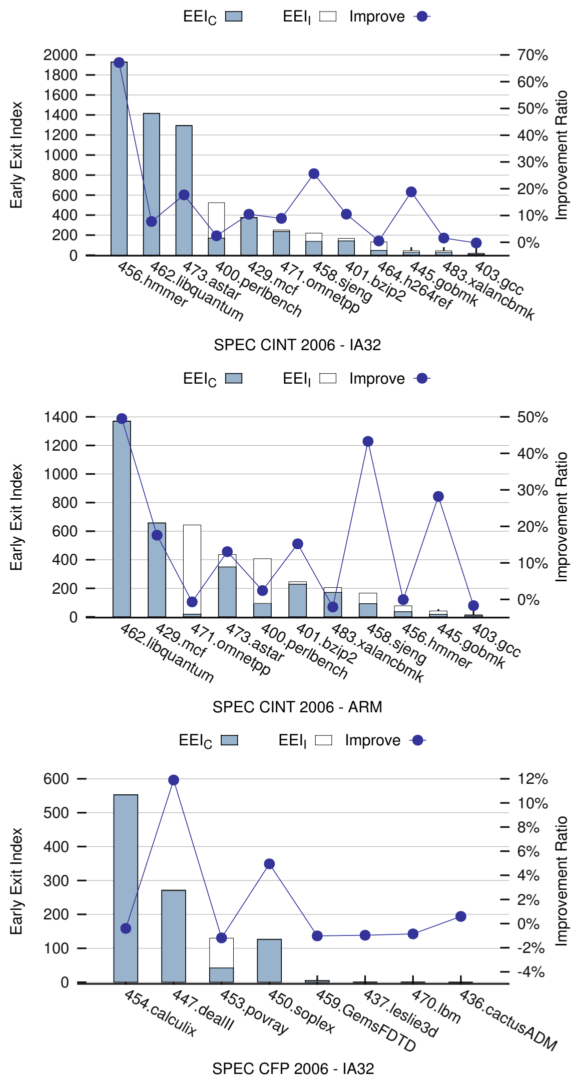

Figure 5: Early Exit Index

region selection come from eliminating the overhead caused by frequently-taken early exits and potential optimization opportunities from a larger region. Benchmarks such as `sjeng` and `gobmk` have small EEI values, but achieve good performance improvements, i.e. they have 43% and 28% performance improvements respectively. The reason is that the LLVM JIT compiler has the opportunities to perform more optimizations with a larger region.

Some benchmarks, such as IA32 `libquantum`, have high EEI values but relatively small performance improvement. The reason is that, althoug EEG reduces significant amount of memory operations, as shown in Table 2, IA32 `libquantum` is a CPU intensive application and therefore, the eliminated execution time from those memory operations has limited impact on overall execution time.

Finally, the `hmmer` benchmark has very different EEI values in IA32 and ARM architectures. The reason lies in the differences of the guest instructions in `P7Viterbi` function of `hmmer` benchmark. In `P7Viterbi`, `hmmer` updates global values according to different conditions in a performance critical `for`-loop. In IA32 `hmmer`, the compiler we used generates a series of compare and jumps instructions. As shown in Figure 1, this `for`-loop is separated into 4 traces by NET. Consequently, the transition among these four traces results in high EEI value. By merging these four traces, EEG successfully form a hot region that contains the performance critical `for`-loop, and thus achieve significant performance improvement – only 1.09X slowdown compared to native execution, as opposed to 1.82X slowdown using the NET scheme.

On the other hand, in ARM `hmmer`, the compiler generates a series of *conditional moves* for that loop. As a result, the loop in ARM `hmmer` has only two basic blocks, which can perfectly be included in a trace. Hence, EEI in ARM `hmmer` becomes very small. That is the reason why EEG region selec-

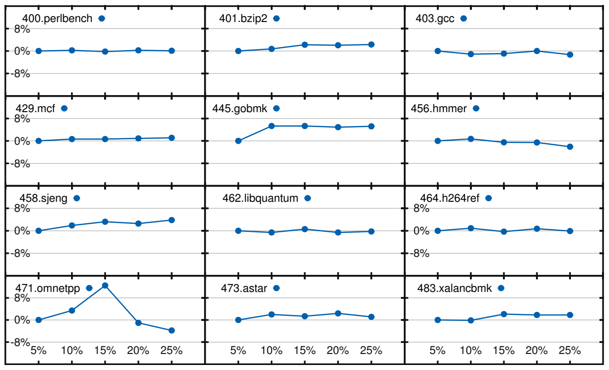

Figure 6: Effect of Spill Index on IA32 SPEC CINT2006.

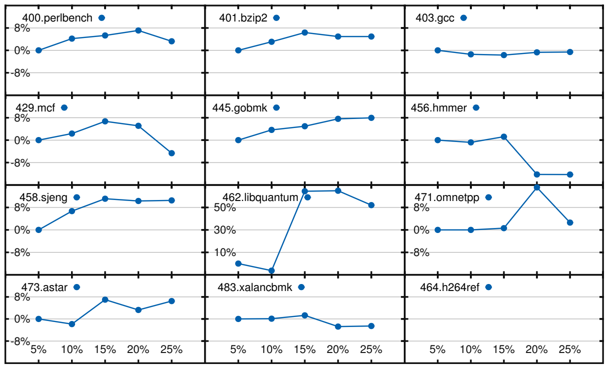

Figure 7: Performance compared to native execution of benchmarks.

tion gain very little performance improvement in this case.

For SPEC CFP2006, all the EEI values are relatively small compared to those in SPEC CINT2006, which indicates that traces in floating benchmarks have fewer early exits than those in integer benchmarks. As a results, 6 floating benchmarks does not gain performance, only `dealII` and `soplex` are improved by 12% and 5% from our EEG region selection.

4.3 Effect of The Threshold of Spill Index

Figure 6 show the effect of the threshold of spill Index on the performance of EEG region selection in IA32 integer benchmarks. Here we use the performance of 5% threshold as the performance baseline. As shown in Figure 6, in 9 out of 12 benchmarks, spill index has no effect on the performance. The main reason is that there are 8 general purpose registers in IA32 and 16 in x86_64. Thus, register pressure may not be a problem when translating IA32 to x86_64.

However, as shown in Figure 7, the performance

Benchmarks	Trace AvgBlk	Region AvgBlk	#Region /#Trace	Avg. Merge
CINT2006 - IA32	2.86	12.35	7.61%	2.15
CINT2006 - ARM	2.51	13.06	7.11%	2.18
CFP2006	3.38	12.09	3.97%	1.76

Table 3: Statistics of Selected Regions

of EEG region selection is more sensitive in ARM integer benchmarks. This is because there are 16 general purpose registers in ARM architecture. Consequently, the LLVM JIT compiler tends to have register pressure issues when translating ARM instructions to x86_64 host. If we allow regions with high spill indices, i.e., high percentage of spill code in the translated code, to be merged, the performance tends to degrade. For example, in ARM `hmmer`, we have 12% degradation when the threshold of spill index increases from 15% to 20%. Similarly, in IA32 `omnetpp`, we have 8% performance degradation when the threshold of spill index increases from 15% to 20%.

4.4 Region Statistics

Table 3 shows the statistics of selected regions. On average, there are 2.51 to 3.38 guest basic blocks in each trace in SPEC CPU2006. Through EEG region selection, we enlarge the size of regions to 12.09 to 13.06 guest basic blocks. Therefore, about 7% and 4% of traces are selected as regions through EEG region selection in integer and floating benchmarks, respectively. The average number of times that a region is merged is 2.18.

4.5 Performance Results of PARSEC

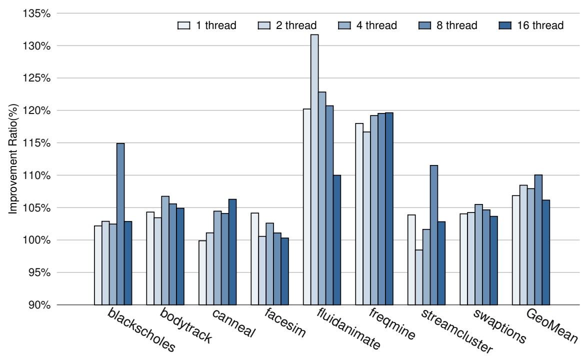

Figure 8: Performance compared to Trace of PARSEC.

Figure 8 shows the performance of EEG region selection on PARSEC multi-threaded benchmarks. EEG achieves 6% to 10% performance improvement compared to Trace configuration on average for the

PARSEC benchmarks . We achieves up to 10% improvement when using 8 guest threads in both configurations. However, the performance drops to 6% when 16 threads are used. Since in both Trace and Region configurations there are 2 optimization threads competing CPU with the execution threads, the degradation may come from the profiling thread in Region configuration. We plan to dynamically adjust the sampling frequency of profiler in our future work.

4.6 Performance Comparison to Native Execution

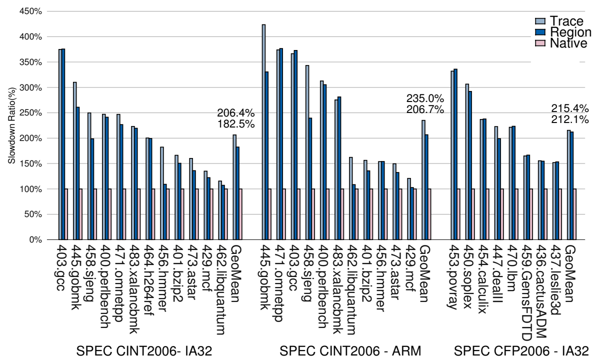

Figure 9: Performance compared to native execution of SPEC CPU2006.

In the last set of experiments, we show the performance of our region-based DBT systems compared to native execution. As shown in Figure 9, our IA32-to-x86_64 dynamic translator has 1.82X, 2.12X slowdown with SPEC CPU2006 integer and floating-point benchmarks respectively, compared to 2.06X, 2.15X slowdown using Trace configuration. For ARM-to-x86_64 DBT, it has only 2.06X slowdown compared to native execution of SPEC CPU2006 integer benchmarks, as opposed to 2.35X slowdown using the NET scheme. Furthermore, our system results in less than 1.5X slowdown for 5 ARM integer benchmarks; in particular, `mcf` and `libquantum` have only 1.02X and 1.08 slowdown ratio compared to native execution on x86_64 host .

As shown in Figure 10, our systems achieve 1.96X to 2.29X slowdown ratio compared to native execution on x86_64 host for PARSEC multi-threaded benchmarks. In Figure 10, the geometric mean of slowdown ratios increases from 2.10X to 2.29X when guest threads increases from 8 to 16 threads. As mentioned above, the overhead of dynamic profiling becomes significant due to CPU competition when all 16 CPU cores are used.

5. RELATED WORKS

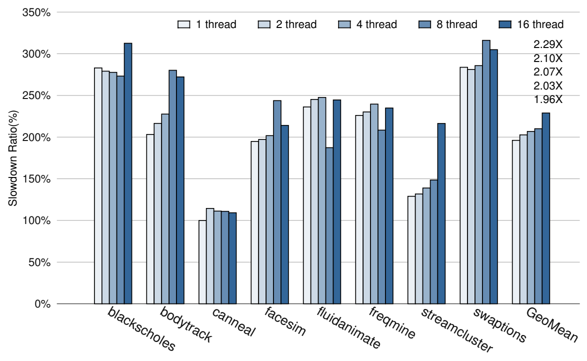

Figure 10: Performance comparison with native execution of PARSEC multi-threaded benchmarks.

We give an overview of the related work, and point out their differences with our approaches. We focus on dynamic binary translation, trace-based JIT compilation and region-based JIT compilation. Dynamo [3] was the first trace-based optimizing compiler that used the Next-Tail-Execution (NET) algorithm. Dynamo pioneered many early concepts of trace selection and trace runtime management. Though NET performs well in practice, as shown in our experiments, NET is likely to select delinquent traces and hence has room for improvement. Hiniker et al. [10] proposed LEI and their trace combination algorithm, which needs to interpret each taken branches to select traces. As pointted out in [11], our approach uses hardware performance monitor sampling technique to efficiently select candidate regions to be merged. Ha et al. [9] and Bohm et al. [6] both propose the strategy of spawning one or multiple optimizations threads for JIT trace compilation so that concurrent interpretation and JIT trace compilation can be achieved. In contrast we use optimization threads to compile not only traces but also regions. COREMU [16], a full-system emulator based on QEMU, emulates multiple cores by creating multiple instances of sequential QEMU emulators. The system is parallelized by assigning multiple QEMU instances to multiple threads.

Gal et al. [8] merges traces via trace trees technique that focuses on loop traces. Our work focuses on all delinquent traces, not limited to loop traces. Their region formation must enter interpreter mode while selecting instructions to merge until terminal conditions meet. While our approach effectively select all blocks in region that starts at the target address of the early exit. In addition, we use multiple helper threads for region compilations which minimize the compilation overheads.

Suganuma et al. [15] studied region-based compilation for JAVA JIT compilation which is to perform partials inlining at runtime. Rather than inlining the entire function, this work studied how to select regions to inline by eliminating rarely executed sections of code of the called function. They dynamically profile the execution counts of basic blocks with instrumentation counters, and use static code analysis of JAVA bytecode to identify rarely executed code such as exception handling. Their approach does not fit our scenario because, unlike JIT compiler in JAVA, it is difficult to identify rarely executed region through static binary code analysis in binary translation.

6. CONCLUSION

In this paper, we propose a light-weight region formation technique called Early-Exit Guided region selection (EEG) to significantly improve the performance of NET by merging delinquent traces into larger code regions. Hardware-assisted dynamic profiling is first used to identify hot code regions without incurring significant runtime overhead. Key software counters are then instrumented at the exit points of the hot regions to detect early exits. When a counter exceeds certain threshold value, the code region that begins at the branch target of that early exit is merged into the main code region. We also employ a heuristic to decide whether it is beneficial to merge the selected regions or not.

We implement our EEG algorithm in two LLVM-based parallel dynamic binary translators. These two parallel dynamic binary translators use multiple compilation threads to compile different code regions concurrently. We evaluate the performance of EEG with two benchmark suites: the SPEC CPU2006 single-threaded benchmark suite with reference inputs, and the PARSEC multi-threaded benchmarks with native inputs. The experimental results show that, compared to NET, EEG can achieve a performance improvement of up to 67% (13% on average) for SPEC CPU2006 integer benchmarks, and up to 20% (10% on average) for PARSEC multi-threaded benchmarks. It also reduces memory operations in a benchmark by up to 76%.

7. REFERENCES

[1] Low Level Virtual Machine (LLVM). http://llvm.org.

[2] QEMU. http://qemu.org.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00*, pages 1–12. ACM, 2000.

[4] 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. In *MICRO-36*, pages 191–201, Dec. 2003.

[5] F. Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[6] I. Bohm, T. E. von Koch, S. Kyle, B. Franke, and N. Topham. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proc. PLDI*, 2011.

[7] D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation.* Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Sep 2004.

[8] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *PLDI*, pages 465–478, 2009.

[9] J. Ha, M. Haghighat, S. Cong, and K. McKinley. A concurrent trace-based just-in-time compiler for single-threaded javascript. In *Workshop on Parallel Execution of Sequential Programs on Multicore Architectures*, 2009.

[10] D. Hiniker, K. Hazelwood, and M. D. Smith. Improving region selection in dynamic optimization systems. In *MICRO 38*, pages 141–154, Washington, DC, USA, 2005. IEEE Computer Society.

[11] D.-Y. Hong, C.-C. Hsu, P. Liu, C.-M. Wang, J.-J. Wu, , P.-C. Yew, and W.-C. Hsu. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *CGO '12: Proceedings of the 10th annual IEEE/ACM international symposium on Code generation and optimization*, 2012.

[12] C.-C. Hsu, P. Liu, C.-M. Wang, J.-J. Wu, D.-Y. Hong, P.-C. Yew, and W.-C. Hsu. Lnq: Building high performance dynamic binary translators with existing compiler backends. In *ICPP*, pages 226–234, 2011.

[13] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *15th Annual ACM Symposium on Principles of Distributed*

Computing, 1996.

[14] perfmon2. http://perfmon2.sourceforge.net.

[15] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a java just-in-time compiler. *SIGPLAN Not.*, 38:312–323, May 2003.

[16] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang. COREMU: a scalable and portable parallel full-system emulator. In *Proc. PPoPP*, 2011.