Processor-Tracing Guided Region Formation in Dynamic Binary Translation

DING-YONG HONG and JAN-JAN WU, Institute of Information Science, Academia Sinica, Taiwan
YU-PING LIU, SHENG-YU FU, and WEI-CHUNG HSU, Department of Computer Science and Information Engineering, National Taiwan University, Taiwan

Region formation is an important step in dynamic binary translation to select hot code regions for translation and optimization. The quality of the formed regions determines the extent of optimizations and thus determines the final execution performance. Moreover, the overall performance is very sensitive to the formation overhead, because region formation can have a non-trivial cost. For addressing the dual issues of region quality and region formation overhead, this article presents a lightweight region formation method guided by processor tracing, e.g., Intel PT. We leverage the branch history information stored in the processor to reconstruct the program execution profile and effectively form high-quality regions with low cost. Furthermore, we present the designs of lightweight hardware performance monitoring sampling and the branch instruction decode cache to minimize region formation overhead. Using ARM64 to x86-64 translations, the experiment results show that our method achieves a performance speedup of up to 1.53× (1.16× on average) for SPEC CPU2006 benchmarks with reference inputs, compared to the well-known software-based trace formation method, Next Executing Tail (NET). The performance results of x86-64 to ARM64 translations also show a speedup of up to 1.25× over NET for CINT2006 benchmarks with reference inputs. The comparison with a relaxed NETPlus region formation method further demonstrates that our method achieves the best performance and lowest compilation overhead.

CCS Concepts: • **General and reference** → **Design**; **Performance**; • **Software and its engineering** → **Dynamic compilers**;

Additional Key Words and Phrases: Dynamic binary translation, region formation, processor tracing, hardware performance monitoring, next executing tail

52

ACM Reference format:
Ding-Yong Hong, Jan-Jan Wu, Yu-Ping Liu, Sheng-Yu Fu, and Wei-Chung Hsu. 2018. Processor-Tracing Guided Region Formation in Dynamic Binary Translation. *ACM Trans. Archit. Code Optim.* 15, 4, Article 52 (November 2018), 25 pages.
https://doi.org/10.1145/3281664

This work is supported in part by Ministry of Science and Technology of Taiwan under grant number MOST-106-2218-E-002-040 and MOST-107-2221-E-001-002.

Authors' addresses: D.-Y. Hong and J.-J. Wu, Institute of Information Science, Academia Sinica, 128 Academia Road, Section 2, Nankang, Taipei 115, Taiwan; emails: {dyhong, wuj}@iis.sinica.edu.tw; Y.-P. Liu, S.-Y. Fu, and W.-C. Hsu, Department of Computer Science and Information Engineering, National Taiwan University, 1, Section 4, Roosevelt Road, Taipei 10617, Taiwan; emails: {r04922005, d03922013, hsuwc}@csie.ntu.edu.tw.

ACM 1544-3566/2018/11-ART52
https://doi.org/10.1145/3281664

1 INTRODUCTION

Dynamic binary translation (DBT) is a virtualization technique that can emulate application binaries of one instruction set architecture (ISA) on a host machine with a different ISA. It operates directly on binaries and stores the translated codes in the code cache to avoid re-translation. DBT has been widely used in many important applications for different purposes, such as dynamic re-optimization, binary instrumentation, security analysis, and cross-ISA application migration. For example, Dynamo (Bala et al. 2000), ADORE (Lu et al. 2004), and StarDBT (Wang et al. 2007) recognize optimization opportunities and improve the execution performance at runtime. Pin (Luk et al. 2005) and Valgrind (Nethercote and Seward 2007) analyze program behaviors with dynamic instrumentation. To migrate applications across ISAs, the IA-32 EL software (Baraz et al. 2003) translates legacy IA-32 binaries into Itanium codes, whereas the Android emulator allows ARM applications to run on x86-based machines, which facilitate Android software development.

Region formation is the important first step in DBT to select frequently running code for translation and optimization. There are two major reasons for selecting hot regions. First, translation and optimization incur overhead to program execution, and thus it avoids compiling cold codes so as to reduce system overhead. Second, focusing on the frequently running codes ideally can gain the most benefit from the optimization. A region can be a basic block, a trace (i.e., a single-entry multiple-exit path, as known as superblock), or as large as a whole function.

Typically, region formation consists of two steps: (1) profiling code segments to detect the "hotness" of regions and (2) selecting code segments to build regions. As regions are forming at the same time the program is running, the overall performance is very sensitive to the formation overhead, because code profiling and selection can have a non-trivial cost. Moreover, the quality of the formed regions determines the extent of optimizations, and thereby also determines the final execution performance. Therefore, how to effectively select high-quality regions and maintain low overhead is an important dual-issue in designing a region formation algorithm.

A common technique to capture the hot regions of a program is *instrumentation*. The methods of basic block profiling and edge profiling (Ball and Larus 1994) find hot execution paths by inserting monitoring code in all basic blocks and control flow edges, respectively, whereas the path profiling method (Ball and Larus 1996) only instruments distinct execution paths. Though these approaches can accurately identify program hotspots and select high-quality regions, they also result in significant profiling overhead. Therefore, they are mostly applied in offline tools for profile-guided compilation and software coverage testing (Graham et al. 1982; Neustifter 2010; Tikir and Hollingsworth 2002).

In contrast to the heavy-instrumentation methods, Next Executing Tail (NET) (Duesterwald and Bala 2000) is a lightweight region formation method used in many DBT systems (Bala et al. 2000; Böhm et al. 2011; Bruening et al. 2003; Chen et al. 2000; D'Antras et al. 2017; Hong et al. 2012). The NET algorithm only profiles basic blocks of potential loop heads, and it captures hot execution paths by iteratively forming traces that follow the potential loop heads (described in Section 2.2). With its simple selection heuristic, the NET algorithm can minimize the profiling cost. However, it also causes the critical problem of *trace separation*, where regions with complex control flows (e.g., nested loops) are split into multiple traces. When trace exits occur frequently, not only the benefit of trace optimization is lost, but extra compensation code needs to be executed. As a result, the region quality decreases. A number of methods have been proposed to prevent early exits by using different trace head candidates or changing the trace termination policies (Castanos et al. 2014; D'Antras et al. 2017; Hayashizaki et al. 2011; Wang et al. 2010). However, these methods still suffer the problem of trace separation.

The requirements of low overhead and high-quality region formation are often in conflict with each other, and usually it is difficult to find the balance when designing a region formation method

based on instrumentation for DBT. In addressing the issues, this article aims at providing a solution that builds high-quality regions but exerts low overhead. To this end, we propose a non-intrusive region formation method guided by *processor tracing*. Nowadays, many modern processors support advanced tracing technology that can record program execution flows (e.g., sources/targets of branches and changes of privilege levels) in processor registers or internal buffers at a low cost and provide programming interfaces for software to retrieve such information. For example, Intel processors support the tracing facilities of LBR, BTS and IPT, and CoreSight for ARM. Based on this technology, we can leverage the *branch history* information recorded by the processor to assist region formation instead of employing instrumentation.

With the help of processor tracing, our processor-tracing guided region formation method has the following advantages compared to the instrumentation-based methods. (1) Program execution profiles, such as hotness and control flows of code segments, can be rebuilt from the branch history. Thus, instrumentation is no longer required for profiling and selecting code segments, and no additional workload is imposed on the execution threads. (2) With the abundant information of the branch history, high-quality regions can be produced. (3) The region formation process can be off-loaded to other threads and does not interrupt the emulation.

The design of the proposed formation method faces several challenging issues. First, whether the processor-tracing event and region formation should be permanently enabled or not? Although the entire region formation workload can be migrated to other helper threads, it makes sense to activate processor tracing and region formation only when they are beneficial, e.g., when code segments become hot. To accomplish this, we design a lightweight hardware performance monitoring (HPM) sampling mechanism that predicts execution phases and enables processor tracing on demand. Second, some processor tracing facilities encode branch records into compressed formats, and a software decoder is required to restore the branch history, which can be extremely expensive (e.g., IPT). To minimize the decoding overhead, we design a lookup table for caching the decoded branch instructions to accelerate the restoration of repeated branches. The key contributions of this article are as follows:

- We design and implement the processor-tracing guided region formation method in HQEMU (Hong et al. 2012), which is a trace-based cross-ISA DBT system. By forming high-quality regions, we improve HQEMU by increasing the optimization opportunities for register mapping and reducing the overhead of guest architecture state synchronization.
- We design the mechanism of lightweight HPM sampling and a branch instruction decode cache, which effectively reduce the overall system overhead.
- We evaluate the translations on the x86-64 host by using three processor-tracing facilities: Intel LBR, BTS, and IPT. The ARM64 to x86-64 translation results indicate that compared with the NET algorithm, a speedup of up to 1.53× (1.16× on average) is achieved through IPT-guided region formation for the SPEC CPU2006 benchmarks with reference inputs.
- We evaluate the translations on the ARM64 host by using the ARM CoreSight tracing facility. The x86-64 to ARM64 translation results show that compared with NET, our method achieves a speedup of up to 1.25× (1.13× on average) for CINT2006 benchmarks with reference inputs.
- We compare our method with a relaxed version of the NETPlus region formation algorithm (Davis and Hazelwood 2011). The performance results indicate that our method achieves 1.06x speedup and takes only 25% compilation time over the relaxed NETPlus method.

The remainder of this article is organized as follows. Section 2 provides an overview of the HQEMU framework and describes the NET selection algorithm and its problem. Section 3 presents

the processor-tracing guided region formation method. We report the evaluation results in Section 4. Section 5 describes related work, and Section 6 concludes.

2 BACKGROUND AND MOTIVATION

Our region formation method is implemented in a NET-based DBT system called HQEMU. Because HQEMU is used as the baseline platform in this work, we begin this section with an overview of the DBT framework and then explain the NET algorithm and motivating problems.

2.1 DBT Infrastructure

HQEMU is a retargetable cross-ISA DBT system. It leverages the frameworks of QEMU (Bellard 2005) and LLVM (Lattner and Adve 2004) as its frontend emulator and backend optimizer, and supports several major ISAs, such as x86-32, x86-64, ARMv7, and ARMv8. To minimize the translation overhead, HQEMU runs stage translation. The guest binary is first translated with moderate optimization by a fast translator (i.e., QEMU TCG). The hot execution codes identified using the NET algorithm are then aggressively optimized through the LLVM optimization passes for better performance. TCG translates the guest binary one basic block at a time and stores the translated codes in a basic block code cache, whereas the LLVM optimizer deals with traces with a trace code cache. To reduce the optimization overhead, HQEMU migrates the workload of LLVM optimizations to other helper threads.

HQEMU uses intermediate representation (IR) to achieve retargetable binary translation across many guest and host ISAs. For guest instruction translations, trivial instructions are translated to IR instructions, which are then translated to equivalent host instructions and stored in the code cache. Complex instructions, such as ARM NEON instructions, are implemented using helper functions, which are pre-compiled to the host binary. At runtime, complex instructions are emulated by calling out the helper functions.

Similarly to many cross-ISA DBTs, HQEMU maintains the guest architecture states (i.e., guest CPU) in the host memory. To minimize memory access to the architecture states, frequently used guest registers are mapped in IR virtual registers, which are in turn mapped to physical registers on the host machine—a process called *register mapping*. If a virtual register is modified, then its content is saved back to the architecture state mapped in memory at the exit points—a process called *architecture state synchronization*. The overhead of state synchronization can be mitigated by increasing translation code granularity. Therefore, HQEMU forms NET traces to combine frequently executed basic blocks, and unnecessary state synchronization is optimized out through guest state promotion.

2.2 NET

NET is a popular trace formation method, which was first developed in the Dynamo DBT system. The NET algorithm is based on a simple concept: when a basic block becomes hot, the following executed basic blocks are likely to be hot as well. Therefore, NET instruments counters for the basic blocks of potential trace heads. When a counter reaches the hot threshold, NET begins trace tail selection by recording the following executed basic blocks until a termination condition is encountered. The basic blocks that can act as trace head candidates are the targets of backward branches or the exit targets of existing traces. A trace tail is terminated if a backward branch is encountered, the next block is a trace head, or maximum trace length is reached.

By using backward branches as indicators to start and stop trace formation, the NET algorithm intends to select loops, which are often program hotspots. The instrumentation cost is minimized because only a limited set of basic blocks is monitored. To further reduce the profiling overhead and obtain traces as early as possible, the hot threshold is often set to a small value (e.g., 50 in Dynamo).

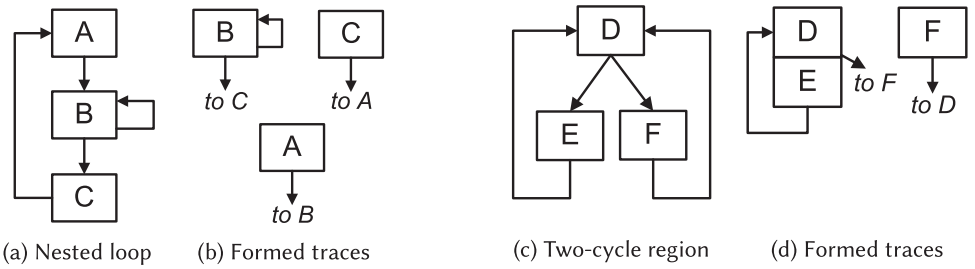


Fig. 1. Examples of NET trace formation. The NET selection algorithm causes the problem of trace separation, where a region with a complex control flow graph is split into multiple sub-regions.

Figure 1 illustrates two NET trace formation examples. Figure 1(a) shows a nested loop, where each block represents a basic block. If, for example, the inner loop block *B* reaches the hot threshold first and the execution continues in *B*'s loop iteration, then block *B* forms a loop trace with a backward branch to itself. Since block *C* is an exit target of trace *B*, it becomes a potential trace head. As *C* becomes hot later, a new trace selection begins by following block *C*. Because the next block, *A*, is a backward branch target, the trace tail construction stops, and a straight-line trace *C* is formed. Block *A* becomes another straight-line trace when it later reaches the hot threshold. In the end, this nested loop produces three separate traces, and their execution flows are depicted in Figure 1(b).

The nested-loop example indicates that the NET selection algorithm has one major flaw: *trace separation*. Such a separation problem occurs because the NET heuristic forms only two trace shapes: (1) simple loops with one backedge from the last block to the loop head block (i.e., O-shape traces) and (2) straight-line code fragments (i.e., I-shape traces). Because of this restriction, a region with a complex control flow graph (CFG) is split into multiple separate traces. This situation is also illustrated in Figure 1(c) and (d), in which the two-cycle region is split.

In cross-ISA DBT systems such as HQEMU, the problem of trace separation can cause critical performance issues if trace exits occur frequently. For example, in Figure 1(d), the execution may often leave the trace *DE* due to unbiased branches. When early exits occur frequently, the benefits of optimizations, such as register mapping for promoting guest states, are lost. Moreover, frequent early exits also result in significant overhead because of the frequent synchronization of guest architecture states.

These issues can be overcome if the complex region is formed in one translation code fragment instead of being split (e.g., constructing the nested loop in one single region, as displayed in Figure 1(a)). As the translation granularity increases from traces to regions, the performance can be enhanced by eliminating the state synchronization overhead and creating additional optimization opportunities in register mapping between the guest and host.

One possible approach for realizing the aforementioned idea is extending the NET algorithm to keep track of a considerable number of executed blocks from the trace head. From this execution history, a better region can be determined to avoid frequent exits. For instance, as block *B* reaches the hot threshold, we can record a sequence of blocks from the execution in the inner loop's iterations to the outer loop and back to the inner loop (e.g., the sequence of *BB…BBCABB…BB…* is recorded). In this manner, we can form the ideal region of blocks *ABC*. However, this approach can incur significant overhead when tracking numerous blocks. Furthermore, it is difficult to determine how long the recording should continue to cover all the blocks of a hot region.

With advanced processor-tracing technology, the execution history can be re-constructed by leveraging the branch records stored in the processor. Therefore, this work is motivated and we propose the region formation method guided by processor tracing.

3 PROCESSOR-TRACING GUIDED REGION FORMATION

HPM units have emerged as an integral part of modern processors, and HPM counters have been extensively used to measure low-level processor events, such as execution cycles, instruction counts, and cache misses. To achieve fine-grain measurement, many processor manufacturers have introduced advanced tracing technology that can track program execution flows, such as sources and targets of branches, execution mode transitions, and changes in processor power states. These execution profiles are recorded in processor registers or internal buffers, and programming interfaces are provided for software to retrieve such information. In this article, we refer to this hardware tracing technology as *processor tracing*. For example, the tracing facilities of LBR, BTS, and IPT are supported by Intel processors, and CoreSight is supported by ARM. The system tools, such as GDB and Linux Perf, have exploited this technology to help users find program bugs and diagnose performance bottlenecks, respectively.

Of the execution flow information, we are most interested in the series of branch sources and targets, and such *branch history* is leveraged to assist region formation. In this section, we first provide an overview of the processor-tracing facilities. We then elaborate on the design details of the proposed region formation method.

3.1 Processor-Tracing Facility

3.1.1 Sampling-Based Processor Tracing. Facilities that support sampling-based tracing include Intel LBR (Last Branch Record), Itanium BTB (Branch Trace Buffer), and PowerPC BHRB (Branch History Rolling Buffer). Typically, this technology provides a limited number of registers to store branch records. The processor traces all the executed branches, but only the most recent branches are logged in the registers (usually 4 to 32 records). To extract branch data from the processor registers, the software configures an HPM sampling event together with a sample period (e.g., one million cycles). Branch records are dumped into the software's buffer each time the sampling event triggers processor interrupt. Because very few branches are saved over the sample period of time, the collected branch history is disjointed and fuzzy with this technology.

3.1.2 Non-Sampling-Based Processor Tracing. Facilities that support non-sampling-based processor tracing include Intel BTS (Branch Trace Store), Intel PT (Intel Processor Trace), and ARM CoreSight. This technology does not require HPM sampling to extract branch data. The processor traces all the executed branches and writes branch records to the software's buffer either directly or indirectly via caching in the internal buffers. The branch records are stored in the order in which they occur; thus, the branch history is continuous.

Intel BTS writes each branch into the software buffer directly. It must clear the instruction pipeline to maintain correct branch ordering, and the branch records are stored in the raw form as branch source/target addresses.

Intel PT (a.k.a. IPT) (Intel Corporation 2018) is the successor of Intel BTS. It improves the tracing performance by caching branch data in the internal buffer. Instead of writing raw branch data in the software buffer, the branch records are encoded into IPT packets. For example, the TNT packet[1] uses one bit to indicate whether the conditional branch is taken or not. A software decoder is required for converting IPT packets back to branches. The execution of direct unconditional branches is not logged by IPT and no packet type representing direct unconditional branches is provided, because these branch information can be derived from walking the program binary. Therefore, the cost to restore the branch history can be extremely high due to the software overhead of decoding

[1]Taken Not-Taken (TNT) packets track the direction of direct conditional branches with 1 for taken and 0 for not taken.

Table 1. Comparision of Processor-tracing Facilities

	Intel LBR	Intel BTS	Intel PT	ARM CoreSight
Tracing mechanism	Sampling	Non-sampling	Non-sampling	Non-sampling
Branch history	Disjointed & fuzzy	Continuous	Continuous	Continuous
IP filtering	No	No	Yes	Yes
Hardware overhead	Depend on sample freq	Medium	Low	Low
Software overhead	No	No	High	High

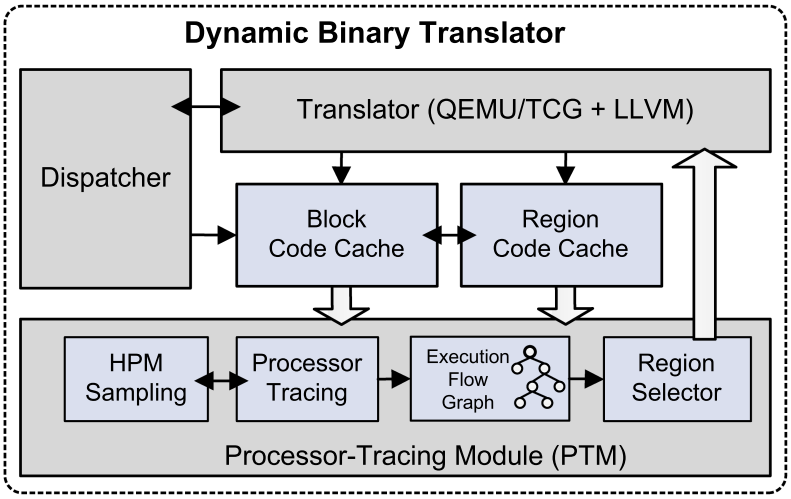


Fig. 2. The DBT framework of the processor-tracing guided region formation.

IPT packets and the program binary. IPT supports hardware instruction point (IP) filtering. It can be configured to record only the branches executed inside or outside specific address ranges.

ARM CoreSight (ARM 2012) is similar to IPT technology. It supports hardware IP filtering, encodes branch data, and requires software decoding.

The comparison of the processor-tracing facilities used in this article is summarized in Table 1.

3.2 Methodology

When processor tracing is enabled, the execution of a guest program is monitored, and the branches executed by the processor are logged. Our goal is to (1) convert the branch records to an execution flow graph of the guest program and (2) determine from the graph the shape of regions that can achieve minimal exits.

Figure 2 illustrates the basic components of the DBT framework extended from HQEMU. A new processor-tracing module (PTM) is added in the framework, which is in charge of the region formation process. Before we can construct the execution flow graph, we must extract branch data from the processor into the memory space allocated by PTM. To do this, a processor-tracing buffer is installed for each emulation thread to receive branch records. For simplicity, we assume that each branch record stored in the tracing buffer is in the raw form of <branch property, branch source address, branch target address>, where the branch property may indicate whether the branch is taken/non-taken, direct/indirect, and so on.

When a sufficient number of branch records are written in the tracing buffer, the branch data are processed to construct the execution flow graph. The execution flow graph is a subset of the guest binary's CFG, in which only the basic blocks executed are included. In the graph, each node maps to one guest basic block, and the edges map to branch links. Moreover, each node and edge is assigned one execution counter. To construct the graph, we follow the execution flow of the traced branch records. A new node/edge is inserted into the graph if the mapped block/branch is encountered for the first time. The execution counter of the node or edge is incremented by one each time the execution enters the associated block or branch. In the end, an execution flow graph that contains the *execution frequencies* of the guest basic blocks and branch links is constructed.

The region selector is then kicked off to conduct region formation. The region formation method proceeds in two steps, which aim at selecting the sub-graphs of the execution flow graph as the resultant regions.

In the profiling step, the region selector determines the "seeds" in the execution flow graph as the starting points for region selection. To do this, we follow the idea of NET to quickly cover program hotspots by beginning region selection from potential loop heads. Thus, a node is set as the region head candidate if it is (1) a target of a backward branch or (2) an exit target of existing regions. When branch data are processed to update the execution flow graph, these branches are also inspected to determine whether any region head candidate exists. If a branch maps to a backward edge in the graph or the source/target addresses are respectively within the range of region/block code cache, then a region head candidate is found in the target node of this branch. Region head candidates whose node frequency exceeds the hot threshold are collected to select region bodies.

In the selection step, the region selector determines the shapes of region bodies for the collected region heads. The region body is constructed by following the outgoing edges from the head node and iteratively selecting the nodes traversed. The question raised here is: how to determine whether a traversed node should be selected? Moreover, poor selection of the region body can degrade the performance through frequent region exits. To solve these issues, we utilize the information of edge frequencies to guide the selection.

Based on the edge frequencies, we compute the *reaching probability* of each node from the region head. The rules for computing the reaching probabilities are as follows:

(1) For each node, the probability of an outgoing edge is the edge frequency divided by the total edge frequency of all the outgoing edges.
(2) The reaching probability of the region head node is 1.
(3) A child node's reaching probability is derived by multiplying the parent node's reaching probability with the linking edge's probability.
(4) If a node has multiple parents, then its reaching probability is the highest reaching probability derived from the parents.

The computation of the reaching probability begins from the region head node and propagates to the child nodes. An example of such a computation is illustrated in Figure 3, where P_i refers to the reaching probability of node i, E_{ij} refers to the edge probability from nodes i to j, and node 0 is the region head.

After the reaching probabilities are assigned, the region body is composed by selecting the nodes (1) with high probabilities, (2) whose node frequency exceeds the hot threshold, and (3) that do not exceed the maximum region length. In this probability-based selection method, the execution is likely to remain within the formed region, which minimizes the number of region exits. Moreover, the shapes of regions that can be constructed by the proposed algorithm are not limited to simple graphs. The algorithm can form general regions, including the nested loops and graphs of complex

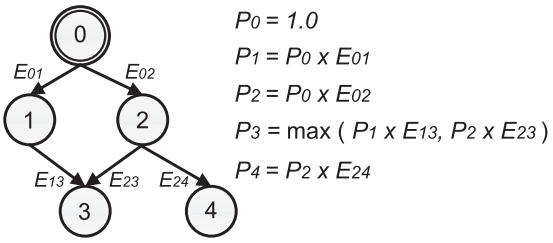


Fig. 3. An example of the reaching probability computation.

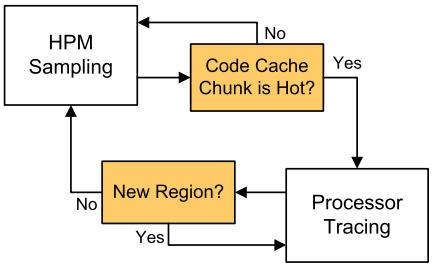


Fig. 4. Controls of HPM sampling and processor tracing.

control flows, as shown in Figure 1(a) and (c). The targets of unbiased branches can also be selected simultaneously if their reaching probabilities are high.

The formed regions are then optimized using the LLVM optimizer. The translated region codes are saved in the region code cache, and the emulation is redirected to the optimized region codes by patching the basic blocks of region heads. Finally, the processor-tracing buffer is flushed so that it can continue to receive new branch records. Execution counters in the execution flow graph are not reset after region formation. The counter values continue to accumulate until the program terminates. Since the DBT system migrates the workload of LLVM optimizations to the helper threads, we assign the job of region formation to one of the threads. This handling thread often stays in the idle state and is only activated when the tracing buffer overflows.

3.3 Lightweight HPM Sampling

In the design described in the previous subsection, the processor-tracing event is enabled at the program startup time and is never disabled. In this situation, the processor-tracing buffer periodically overflows and triggers the update of execution flow graph (software decoding may be run to restore branch history, e.g., IPT) and the profiling and selection steps to form regions. However, the region selector does not always produce new regions if no basic block becomes hot in the recent tracing period or if the region is already formed. Such a situation can occur when the emulation thread spends a large amount of the recent time running (1) outside the code cache, (2) in the cold execution path of the guest program, or (3) in the translated region codes. Especially after most regions of a program are translated, the emulation thread is likely to execute in the third condition. Although the workload of region formation is handled by another thread and does not delay the emulation, the permanent enabling of processor tracing is unacceptable, because processor time and energy are wasted when no region is produced.

The ideal design is to perform processor tracing and region formation on demand. That is, they are disabled if the aforementioned conditions occur and enabled only when a hot and untranslated region appears. Therefore, the condition that the emulation thread runs into must be determined to design such a demand-based scheme. One possible approach to detect the condition is to instrument monitoring codes in the DBT components, including the code caches and dispatcher. However, this approach can incur significant instrumentation overhead. Another possible approach is to permanently enable processor tracing but region selection is skipped if any of the aforementioned conditions is recognized from the branch records. Although this approach optimizes the region formation process, it can also incur significant overhead if the processor tracing requires software decoding to restore the branch records.

Instead, we propose a lightweight HPM sampling mechanism, which predicts the appearance of hot regions on the basis of the captured IPs of the emulation thread. To realize this sampling

mechanism, an HPM sampling event is set up with a sample period. One buffer is installed for each emulation thread to receive the sampled IPs. Moreover, we partition the code cache space into small chunks (e.g., the chunk size is 256 bytes in this work). Each chunk is associated with a profiling counter, which is used to estimate the hotness of the chunk. When the HPM sampling buffer overflows, each sampled IP is verified with the address ranges of each code cache chunk, and the counter of the matched chunk is incremented by one. If a chunk's counter value reaches the hot threshold, then it implies that the code in this chunk is frequently executed. Thus, a potential hot region is predicted.

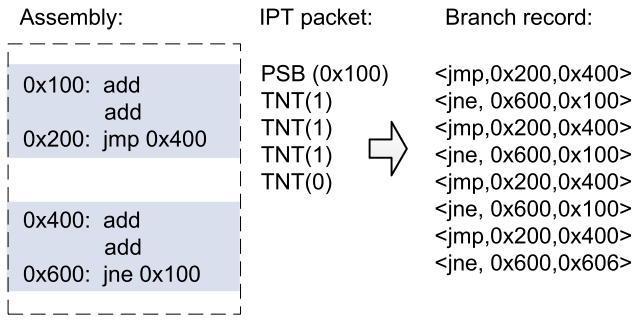
Figure 4 shows the flows between HPM sampling and processor tracing. Initially, HPM sampling is performed to predict the occurrence of potential hot regions. When any chunk becomes hot, the HPM sampling event is disabled, and processor tracing is activated for collecting branch records. Processor tracing is in an active state if the region selector can continue to produce new regions. When no new regions are generated, the processor tracing event is stopped, and the execution returns to HPM sampling to predict the next hot regions. Through this mechanism, processor tracing is only enabled when required. When most regions of a program are generated, the handling thread will mostly remain in the lightweight HPM sampling state. As a consequence, the overhead of region formation is minimized.

The combination of HPM sampling and processor tracing can be beneficial for systems using sampling-based processor-tracing facilities (e.g., LBR). Recall that the branch history generated by the sampling-based tracing facilities is disjointed and fuzzy (Section 3.1.1). To mitigate this problem, the branch history quality can be enhanced by increasing the sampling frequency for processor tracing. However, this enhancement also increases the tracing overhead. Previous research demonstrated that when tracing is enabled permanently, the program performance can degrade significantly with a high sampling frequency (Lu et al. 2004). Because processor tracing is selectively enabled based on our mechanism, we can improve the branch history quality and limit the overall overhead by configuring sampling-based tracing with a burst sampling mode. Therefore, better quality regions can be obtained.

3.4 Branch Instruction Decode Cache for IPT

Processor-tracing facilities, such as IPT and ARM CoreSight, encode branch records in the form of packets, and software decoding is required to convert the packets back into the raw data of branch source/target addresses. In this subsection, we first provide an example that demonstrates the process of software decoding and then present the proposed branch instruction decode cache, which can effectively reduce the overhead of software decoding.

Figure 5(a) depicts a decoding example with IPT. The assembly code is a loop that comprises two executable code segments, 0x100–0x200 and 0x400–0x600. Assume that the loop executes four iterations. The generated IPT packets and restored branch records are listed in the figure (the packet payload is simplified for explanation). Software decoding starts from the first packet, PSB(0x100), which indicates that the processor begins tracing at the address 0x100. Then, the decoder begins disassembling the program binary from this address until a branch instruction is encountered (0x200), and the first branch record, <jmp,0x200,0x400>, is decoded. Since this restored branch is a direct jump, the decoder continues from the jump target address and runs a new disassembling range (0x400–0x600). Because the last instruction is a conditional jump, `jne`, the decoder decodes the next packet, TNT(1), which indicates that the jump is taken. Thus, the jump target address, 0x100, is resolved and the second branch record <jne,0x600,0x100> is restored. Again, the decoder goes to the branch target address and begins the next disassembling range. Such program binary disassembling proceeds repeatedly until all packets are processed.



(a) IPT packets and the branch records decoded for a loop. PSB: packet stream boundary. TNT: taken(1)/non-taken(0) branch.

Key	Value
0x100	<jmp,0x200,0x400>
0x400	<jne, 0x600,X>

(b) Branch instruction decode cache

Fig. 5. Example of a loop and branch record restoration with the branch instruction decode cache. The format of a branch record is <branch property, branch source address, branch target address>.

As illustrated in the example, direct jumps are not recorded in the tracing buffer and no address information is attached in the TNT packets for conditional branches, because such information can be resolved by walking the program binary. Although this design requires considerably less memory usage and achieves a better hardware tracing performance compared with other processor-tracing facilities, the overhead of the software decoder is too high for restoring the branch history. Furthermore, *all* instructions, including the non-branch ones, are disassembled to find the branch instructions. In the aforementioned loop example, all the instructions within the two code segments are disassembled, and the disassembling is repeated for four loop iterations.

Therefore, the restoration of branch records by the decoder can be viewed as a process of collecting the branch instructions (i.e., the last instruction) of the instruction ranges (i.e., disassembling ranges) in the execution order.

Since the aim is to collect branches, the restoration process can be accelerated if the branch instruction can be quickly extracted from an instruction range. The idea is that the branch instruction is remembered after an instruction range is disassembled. Thus, when the same instruction range is entered at another time, the branch can be returned without disassembling the binary again. Therefore, we design a branch instruction decode cache in the IPT decoder to cache the decoded branch instructions. The cache is implemented as a hashtable. The key of the hashtable is the start address of an instruction range and the value is the branch record. Note that the target address of the branch record may not be populated, because IPT packet lookup may be required to resolve it (e.g., TNT for conditional branches). When the decoder begins an instruction range, it first looks up the hashtable according to the range start address. On a hashtable miss, the decoder decodes all the instructions until the next branch instruction is found, and the restored branch record is saved in the hashtable. On a hashtable hit, the cached branch record is returned, and disassembling of the instruction range is skipped.

Consider the code in Figure 5(a) as an example. In the first loop iteration, the decoder must disassemble all the instructions of the two instruction ranges, because the ranges are encountered for the first time. Then, two records are inserted into the hashtable, as displayed in Figure 5(b). In the following loop iterations, disassembling is not required. The branch records are retrieved directly from the hashtable, and the branch target of `jne` is resolved from the TNT packet and the jump instruction's operands.

With the branch instruction decode cache, the restoration of repeatedly executed branches can be accelerated. This scheme is important for our region formation method, because we intend to

build hot regions in which the code segments are very likely to be executed frequently. Thus, the time required for restoring the branch history can be significantly reduced.

4 PERFORMANCE EVALUATION

Our region formation method was implemented in the retargetable DBT system based on HQEMU (QEMU-2.5 + LLVM-6.0). We evaluated the performance with four types of cross-ISA translations: ARM32/ARM64 to x86-64 translations and x86-32/x86-64 to ARM64 translations. The experiments were performed on two host platforms:

- **x86-64 host:** Intel Skylake Core i7-6700 quad-core CPU at 3.40GHz, which supports LBR, BTS and IPT. The host machine has 32GB of main memory and the operating system is 64-bit CentOS 7.2 with Linux kernel 4.11.
- **ARM64 host:** ARM Juno r0 development board with big.LITTLE architecture, which supports the CoreSight tracing facility. The big cluster contains one ARM Cortex-A57 out-of-order dual-core CPU at 1.1GHz, and the little cluster contains one ARM Cortex-A53 in-order quad-core CPU at 850MHz. The host machine has 8GB of main memory and the operating system is 64-bit Ubuntu 16.04 with Linux kernel 4.11.

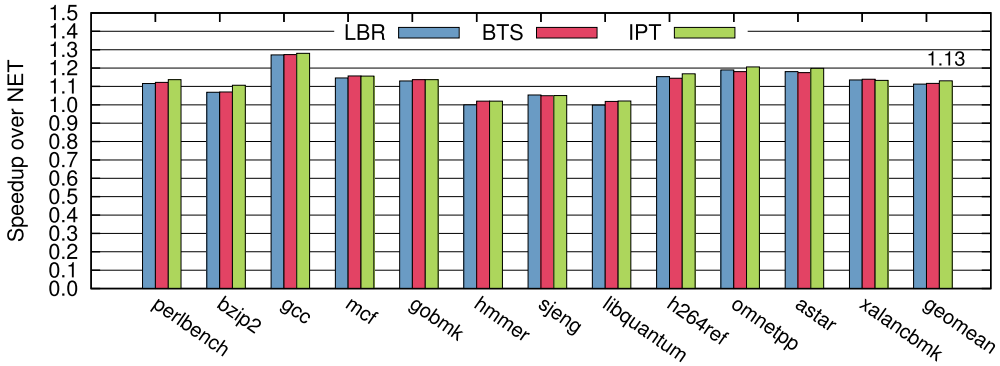
The tracing buffer has a size of 128KB for LBR and BTS, and 4KB for IPT and ARM CoreSight. We allocate more buffer size for LBR and BTS, because their branch data are not encoded to compressed formats. This 128KB tracing buffer can store approximately 5,000 branch records. We set up LBR in a burst sampling mode. The sample period is 1,000 instructions and 32 branches are recorded at every interrupt. For HPM sampling, the sample period is set to 1 million instructions, and the buffer size is 16KB. The branch instruction decode cache comprises 4,096 entries. We use Linux Perf APIs to implement processor tracing and HPM sampling. The OpenCSD library (Linaro 2018) is used for decoding ARM CoreSight packets.

The SPEC CPU2006 benchmark suite is evaluated with reference inputs in the experiments. We use arm-gcc-5.4 (Linaro ToolChain 2017) and flags "*-O3 -marm –mtune=cortex-a8 -mfpu=neon -ffastmath -ftree-vectorize -static*" for ARM32 guest binary, and flags "*-O3 -ffast-math -ftree-vectorize -static*" for ARM64 guest. For x86-32/x86-64 guest binary, we use gcc-5.4 and flags "*-O3 -m32/-m64 -fno-strict-aliasing -msse2 -mfpmath=sse -ffast-math -static.*" For comparison, the NET trace formation method is used as the performance baseline. We compare the execution time, number of memory instructions executed, and structure of the formed traces and regions. For NET trace formation, the trace profiling threshold is set to 50. For region formation, the hot threshold for the node frequencies is set to 32, and the threshold for the reaching probabilities is 0.2. The maximum length of a NET trace and formed region is 64 basic blocks. One helper thread is used for handling the LLVM optimizations and region formation.

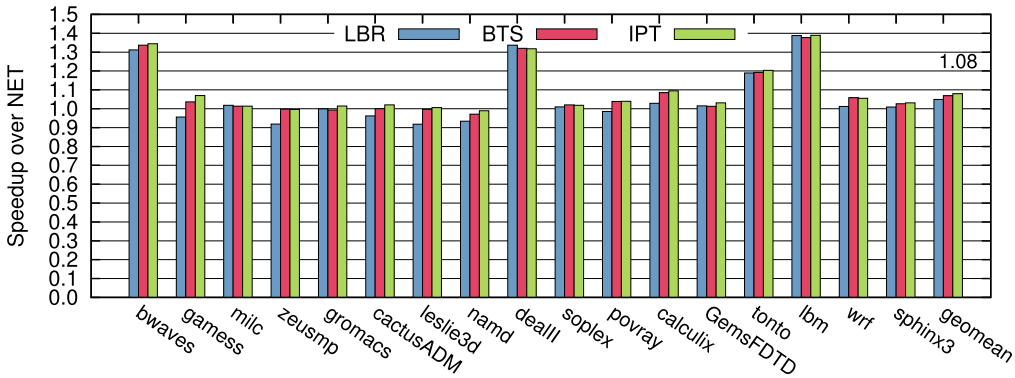
4.1 Performance Results with the ARM32 to x86-64 Translations

Figure 6 shows the performance results of ARM32 to x86-64 translations. The y-axis represents the speedup of our region formation method over the NET method. There exist three bars in each benchmark. The blue, red, and green bars represent the results of LBR, BTS, and IPT, respectively. Benchmark time reported by SPEC is used, and all runtime overhead is included in the measurement.

Overall, IPT-guided region formation achieves the best performance, with a gain of 1.13x in geometric mean for integer benchmarks and 1.08x for floating-point benchmarks compared with NET. The performance of LBR and BTS is slightly lower than that of IPT. On average, our method achieves speedups of 1.08×, 1.09×, and 1.10× for the CPU2006 benchmarks with reference inputs based on LBR, BTS, and IPT, respectively.



(a) CINT2006



(b) CFP2006

Fig. 6. Performance of the ARM32 to x86-64 translations compared with the NET method.

The improvement that can be accomplished with the proposed region formation method is affected by the following factors. (1) The complexity of control flows in the program: Regions of complex control flows are split into traces by the NET formation algorithm, and thus, the more complex the hot regions, the more optimization opportunities can be obtained with our method by reconstructing regions from the separated traces. (2) The frequency of early exits: If few exits occur in the formed traces, there exists limited room for our method to optimize. (3) Operations of the guest architecture states: The overhead of guest state synchronization at the exit points depends on the number of states modified in the traces. If many states are modified, then superior performance can be achieved by eliminating overhead via region formation. Moreover, the use of additional states in the formed regions can create additional optimization opportunities for selecting the best register mappings between the guest and host.

Among the benchmarks, `gcc` and `lbm` have the most significant improvement for the integer and floating-point benchmarks, respectively achieving speedups of 1.28× and 1.38×. Many integer benchmarks achieve remarkable performance gains (>1.10x) with all the processor-tracing facilities. However, only four floating-point benchmarks, i.e., `bwaves`, `dealII`, `tonto`, and `lbm`, achieve significant improvement, and no noticeable performance gain is observed in the other benchmarks. The reason for the difference in performance is that the integer benchmarks have considerably more complex control flows than the float ones. Therefore, integer benchmarks suffer from the severe problems of trace separation and overhead of early exits with the NET method. In contrast, our

method successfully reforms the original complex regions and effectively maintains the execution in the regions. Thus, our method outperforms NET for most integer benchmarks. The same situation is also observed for the four improved floating-point benchmarks. For the benchmarks that exhibit little performance gain, the NET formation method has performed well and few exits are incurred. Therefore, little room for improvement exists on these benchmarks by using our method.

The performance of several floating-point benchmarks (e.g., `gamess` and `zeusmp`) degrades with the LBR-guided method, because the branch history generated by LBR is fuzzy despite LBR being enabled with a burst sampling mode of the highest available frequency (i.e., 1 interrupt per 1,000 instructions). As a result, the execution flow graph is divided into disconnected sub-graphs, and the node and edge frequencies are not as accurate as those of BTS and IPT. The LBR-guided method does not select the best regions. Region exits occur in the formed regions, and the performance of LBR is worse than that of NET for these benchmarks.

Because `lbm` achieves the best performance gain of all the benchmarks, we investigate this benchmark in more detail and compare the results of trace and region formation. `lbm` spends more than 95% of the execution time running in a for-loop in a function named *LBM_performStreamCollide*. The control flow of the for-loop is a two-cycle region (Figure 1(c)), and the execution frequently transitions among three code segments. The computation in the loop of the compiled ARM32 guest binary uses 13 general-purpose registers and 31 VFP registers (*d* registers). When running with NET, this loop is split into two traces, a loop trace and a straight-line trace (Figure 1(d)). Furthermore, early exits occur frequently in the middle of the loop trace. Consequently, huge overhead is incurred, because numerous guest architecture states are frequently stored and reloaded during the transitions between the two separate traces. In contrast, the three code segments are selected in one region in processor-tracing guided region formation. The guest architecture states are effectively kept in the host registers, which results in the execution of 20% fewer memory operations and 38% improvement in the performance compared with NET.

4.1.1 Reduction of Memory Operations. In this subsection, we evaluate the performance of NET and our method by comparing the number of memory instructions executed for the benchmarks with ARM32 to x86-64 translations. The number of memory operations decreases with register mapping when enlarging the translation scope and increases if many trace/region exits occur. Thus, this metric can be used to evaluate the quality of the formed traces and regions. A low number indicates good formation quality. We use HPM counters to measure the number of memory operations executed. For comparison, we use the NET results as a baseline and report the memory reduction rate, which is calculated as follows:

$$\text{ReductionRate} = \frac{\text{MemoryInsn}_{NET} - \text{MemoryInsn}_{PT}}{\text{TotalInsn}_{NET}},$$

where MemoryInsn_{NET} and MemoryInsn_{PT}, respectively, refer to the number of memory operations executed with NET and our method, and TotalInsn_{NET} refers to the total instruction count with NET. A high reduction rate indicates that fewer memory operations are executed in the formed regions than in the NET traces.

Figure 7 illustrates the measurement results. Note that the reduction rate may not exhibit direct correlations with the improvement in the execution performance of the benchmarks, because of the different composition of instructions in the benchmarks, cache effect, and effect of out-of-order execution. For the benchmarks in Figure 6 that exhibit a remarkable speedup, our method also achieves significant memory reduction rates. These results imply that the proposed processor-tracing guided method effectively forms regions with fewer early exits than those of NET. As expected, for the benchmarks that have performance degradation with LBR, the number of executed memory operations is higher than that with the NET method.

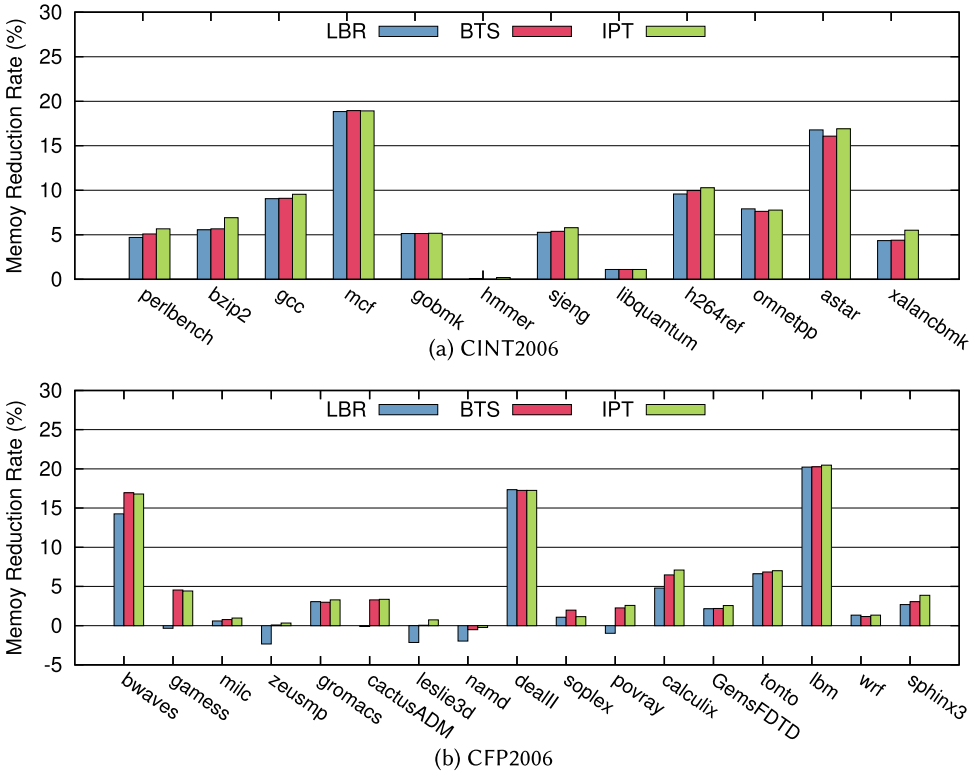


Fig. 7. The reduction of memory operations compared with NET.

Table 2. Number of Traces Generated with the NET Method

	perlbench	bzip2	gcc	mcf	gobmk	hmmer	sjeng	libquan	h264	omnetpp
#traces	10783	6617	78421	396	63728	594	2041	159	7569	937
	astar	xalanc	bwaves	gamess	milc	zeusmp	gromacs	cactus	leslie3d	namd
#traces	913	2831	306	12376	508	1600	884	611	635	1234
	dealII	soplex	povray	calculix	Gems	tonto	lbm	wrf	sphinx3	
#traces	3016	2793	1873	3346	1451	3699	116	5936	2161	

4.1.2 Number of Regions. We measure the number of traces and regions formed with NET and our method. Table 2 lists the number of traces generated by NET for each benchmark, and Figure 8 shows the number of regions normalized to NET. As indicated by the results, our method generates fewer regions than NET with all tracing facilities. This is due to two reasons. First, NET causes trace separation; however, our method combines separated traces back into complex regions. Second, the instrumentation-based NET method captures the traces immediately after the trace head blocks reach the hot threshold (i.e., 50). In contrast, our region selection begins with HPM sampling to detect the occurrence of program hotspots followed by processor tracing to record the branches. Thus, a delay time is involved; therefore, a region must be really hot to be captured. Modestly hot code segments that are selected by NET may not be detected with our method. Fewer regions are obtained with our method, which may result in the loss of some improvement opportunities.

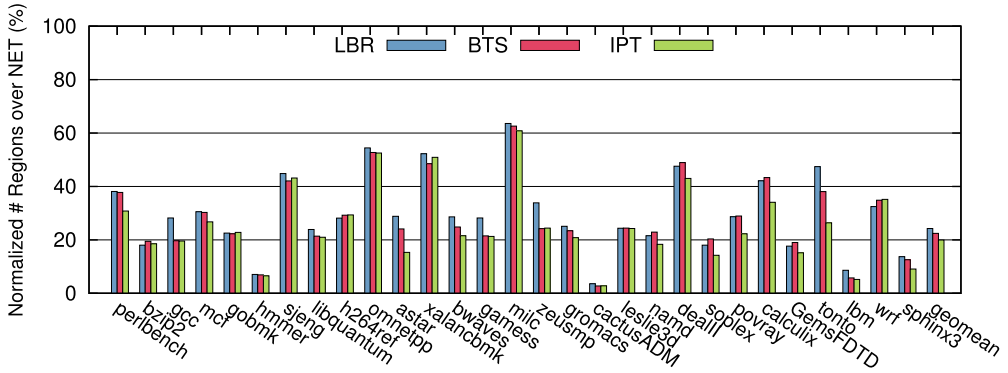
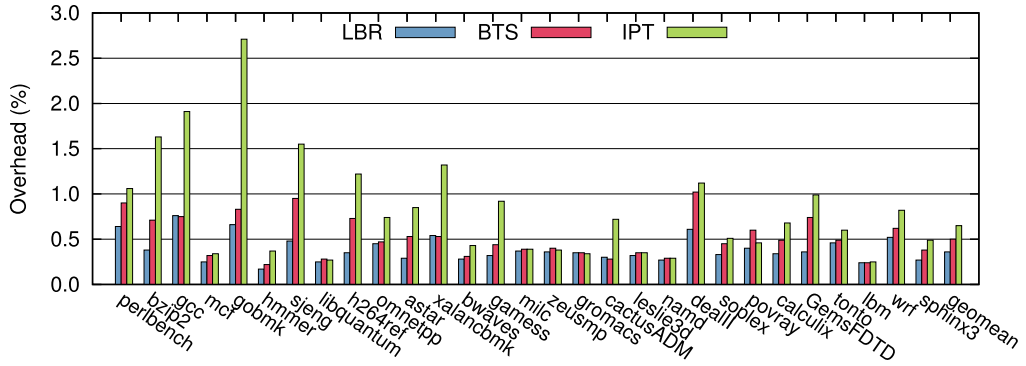


Fig. 8. Number of regions generated.

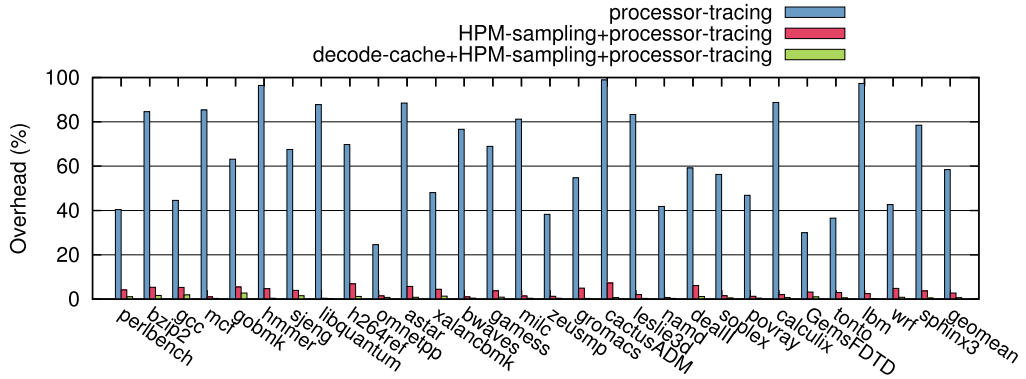
However, it is more important that the separate traces of frequent exits are combined as one, which is the main reason why our method outperforms NET. The LBR-guided method generates more regions than the BTS- and IPT-guided methods for many benchmarks, because some regions generated by BTS and IPT are formed in sub-regions due to the disjointed branch history of LBR.

4.1.3 Overhead of Region Formation. In this subsection, we discuss the overhead of our region formation method. The overhead is measured as the time spent on the region formation handling thread over the benchmark time, which includes (1) the time for performing HPM sampling and (2) the time for processor tracing, which begins from the tracing buffer overflow and ends when the region is formed. The time required for LLVM optimizations and code emission is not included. Figure 9(a) shows the overhead incurred by the three Intel tracing facilities. As the result shows, LBR and BTS incur extremely low overhead (<1%). The overhead of IPT is higher than that of LBR and BTS because software decoding is required for restoring the branch records, which increases the process time. However, the IPT overhead remains within an acceptable range (<3% of the benchmark time). Moreover, the result shows that the integer benchmarks incur more overhead than the floating-point benchmarks, which also implies that the control flows of the integer benchmarks are more complicated than the float ones. More branch records are processed to cover the hot regions of the integer programs.

To evaluate the effects of HPM sampling and the branch instruction decode cache, we disable these optimizations and measure the resultant overhead on the handling thread. Figure 9(b) shows the measurement results. We only report the results of IPT because it is the only Intel tracing facility that requires software decoding. In Figure 9(b), the blue bar represents the results when running processor tracing permanently (no optimization), the red bar represents the results when HPM sampling is enabled, and the green bar is for both optimizations enabled (the same results as the green bar in Figure 9(a)). The results show that permanently enabling processor tracing imposes significant overhead on the handling thread (25–98%). However, as the results in Figure 9(a) indicate, 3% of the execution time is sufficient to detect and form all the hot regions for each benchmark. Therefore, most of the time is wasted by the handling thread when using processor tracing only. By performing the optimization of lightweight HPM sampling, processor tracing is only activated when required. Significant workload of the handling thread is eliminated, and the overhead on the thread is reduced to 1–8%. By adding the optimization of the branch instruction decode cache, the overhead is further minimized to less than 3%.



(a) Region formation overhead with the Intel tracing facilities



(b) Effects of HPM sampling and the branch instruction decode cache with IPT.

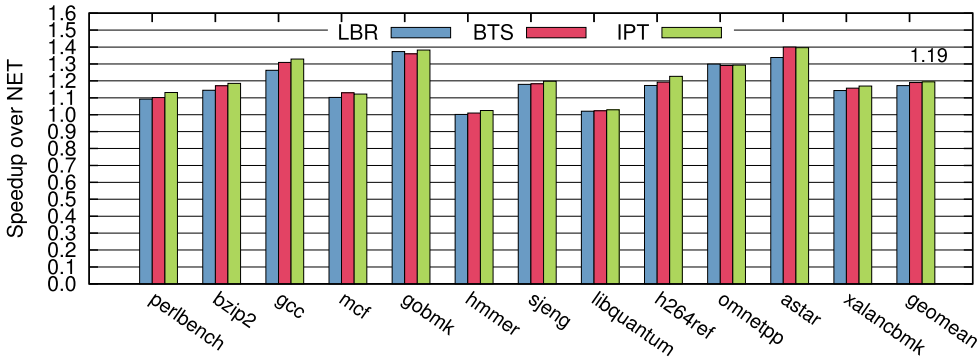
Fig. 9. Region formation overhead.

4.2 Performance Results with the ARM64 to x86-64 Translations

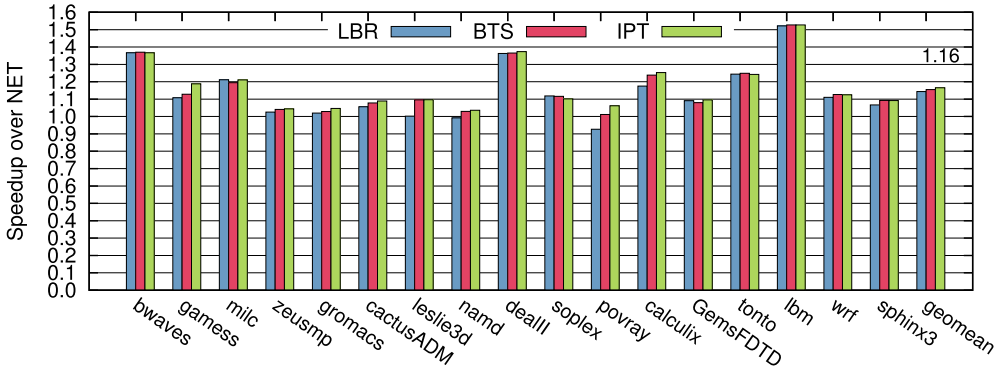
Figure 10 shows the performance and region formation overhead of the ARM64 to x86-64 translations. Similarly to the ARM32 results in Figure 6, our method also achieves significant improvement over the NET method for most integer benchmarks and some floating-point benchmarks, respectively achieving an average speedup of 1.19× and 1.16×. Many benchmarks, such as `gobmk`, `astar`, and `lbm`, show higher speedup with the ARM64 to x86-64 translations than with the ARM32 translations. The reason is because ARM64 comprises more architecture states than ARM32 (32 general-purpose registers and 32 SIMD registers in ARM64 compared with 16 general-purpose registers and 16 SIMD registers in ARM32). The ARM64 program binaries use more registers than ARM32 binaries, and thus additional state synchronization overhead can be eliminated by forming regions. Overall, our method achieves an average speedup of 1.16× for the CPU2006 benchmarks with the three tracing facilities, where less than 3% region formation overhead is incurred (Figure 10(c)). Furthermore, our method achieves 8% fewer memory operations executed and generates fewer regions (20%) on average compared with NET.

4.3 Performance Results with the x86-32 and x86-64 to ARM64 Translations

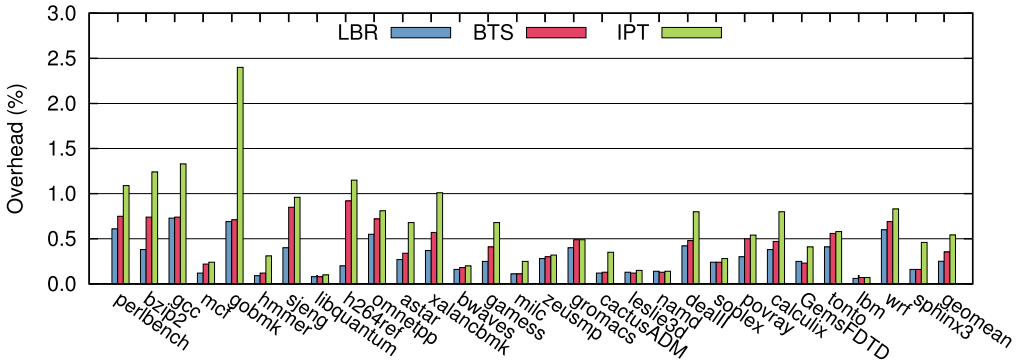
In this experiment, we evaluate the performance of our method based on the ARM CoreSight technology with two translations: x86-32 to ARM64 and x86-64 to ARM64. The emulation and



(a) CINT2006 performance results



(b) CFP2006 performance results

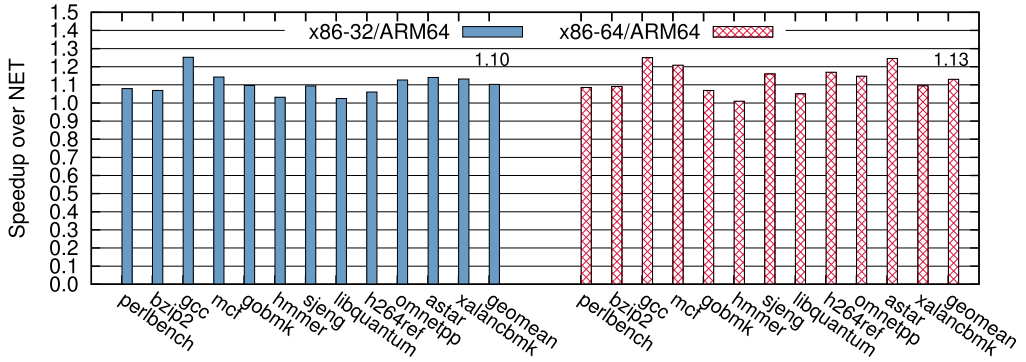


(c) Region formation overhead

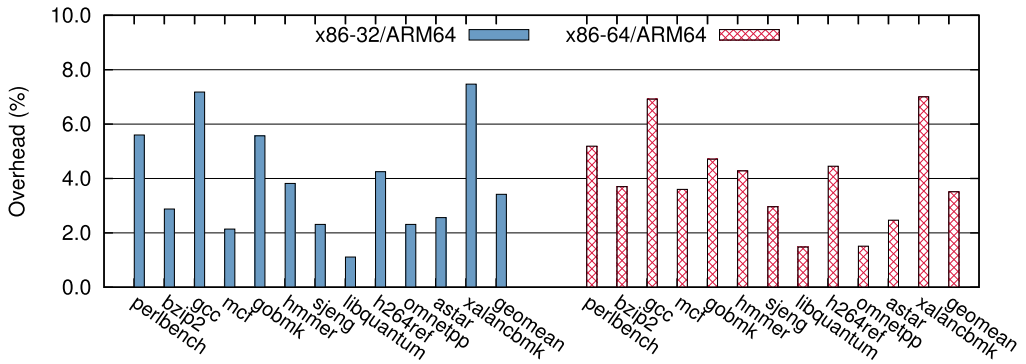
Fig. 10. Performance and overhead of the ARM64 to x86-64 translations.

region formation threads are bound to the two ARM Cortex-A57 out-of-order big cores. The SPEC CINT2006 benchmarks with reference inputs are used for evaluation. Because the Cortex-A57 CPU lacks the performance counter for measuring memory operations, we only report the execution performance and region formation overhead.

Figure 11(a) shows the performance results. On average, our region formation method achieves speedups of 1.10× and 1.13× over NET for the x86-32 and x86-64 translations, respectively. Among



(a) Performance



(b) Region formation overhead

Fig. 11. Performance and overhead of the x86-32 and x86-64 to ARM64 translations.

the benchmarks, `gcc` exhibits the most significant improvement, achieving a speedup of 1.25×. Many other benchmarks, such as `mcf`, `gobmk`, `omnetpp`, `astar` and `xalancbmk`, also show remarkable performance gain (>1.10×). This result is expected because the integer benchmarks have complicated control flows, and our method constructs complex regions instead of separate traces. Therefore, better performance is achieved over the NET trace formation method.

Figure 11(b) shows the overhead of region formation, which ranges from 1.1% (`libquantum`) to 7.5% (`xalancbmk`). The overhead in ARM CoreSight is higher than that in IPT because the processing power of the ARM Cortex-A57 CPU is not as efficient as that of the Intel Skylake processor. Therefore, more time is spent for decoding the CoreSight packets and performing region selection.

4.4 Comparison with the NETPlus Region Formation Method

In the preceding sections, we have presented the importance of region formation for minimizing the architecture state synchronization overhead. Unlike our method, Davis and Hazelwood (2011) proposed NETPlus, which is a software-based region formation method. NETPlus is an extension of the NET trace selection method and comprises two steps. In the first step, the conventional NET algorithm is applied to select a trace. In the second step, a forward search is performed from all exits of the trace for any possible paths that return to the trace head block. A region is formed by combining the NET trace and the basic blocks along the paths to the trace head. We implemented a relaxed NETPlus algorithm in our DBT system, in which the target of the search paths is not

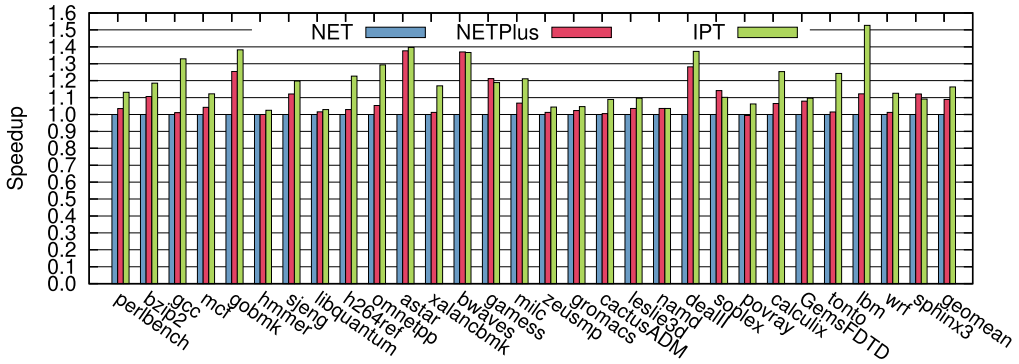


Fig. 12. Performance of NETPlus and IPT-guided region formation with the ARM64 to x86-64 translations.

limited to the trace head block; any path flowing back to any trace body block is included in the region. Such relaxation enables NETPlus to cover additional region shapes, including the if-then and if-then-else control flows that are supported by the IA-32 EL DBT system. In the following, we compare the effectiveness of the relaxed NETPlus and IPT-guided region formation methods. The ARM64 to x86-64 translations are used for evaluation.

Figure 12 shows the speedup of NETPlus and our method over NET. As the result shows, NETPlus achieves performance gain for many programs by avoiding the trace separation problem. The performance of NETPlus is competitive with that of our method for some benchmarks, such as `astar`, `bwaves`, and `dealII`. However, the IPT-guided method outperforms NETPlus for most of the benchmarks. This performance gap is related to the timing of a region's component block selection. NETPlus decides and combines additional paths into a region immediately after the NET trace is completed. Potential paths that contain soon-to-be-executed untranslated blocks are not combined with the NETPlus algorithm. This results in such potential paths being formed as different translation code fragments, which causes separation. In contrast, we select regions on the basis of sufficiently long execution history tracked by the tracing hardware. Therefore, our method could combine all the potential paths in a complete region and minimize region separation. Compared with NET, the overall performance improves by 9% with NETPlus and 16% with our IPT method.

Because enlarging the translation granularity can increase the translation overhead and the traces and regions are aggressively optimized with LLVM passes, we measure the LLVM compilation overhead of each formation method. Figure 13 shows the total compilation time normalized to that of NET. Overall, the NETPlus algorithm spends 28% additional time for LLVM compilation than NET, however, 67% less time is required with the IPT-guided method. Two factors determine the compilation overhead: (1) the region size and (2) number of regions. Although NETPlus generates fewer regions than NET (i.e., 85% the number of NET traces), the total compilation time increases due to the considerably increased overhead for optimizing large regions. In contrast, our method generates considerably fewer regions than NET (i.e., 20% the number of NET traces) because only the really hot regions are selected for optimization. Therefore, the compilation overhead can be significantly reduced.

In summary, we evaluate our processor-tracing guided region formation method with four types of cross-ISA translations. The benchmark results indicate that our method effectively forms high-quality regions with fewer memory accesses and achieves superior performance over NET. Moreover, using lightweight HPM sampling and the branch instruction decode cache significantly

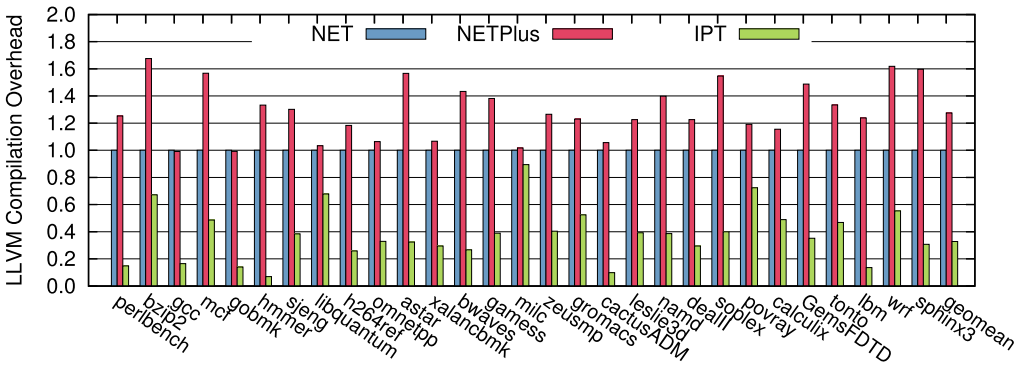


Fig. 13. LLVM compilation overhead with the ARM64 to x86-64 translations.

reduces the system overhead. The comparison with NETPlus also indicates that our method achieves the best performance and lowest compilation overhead.

5 RELATED WORK

After Duesterwald and Bala (2000) proposed the NET selection algorithm in 2000, NET quickly became a popular trace formation method because of its good performance and simple design. Many dynamic compilation systems (Bala et al. 2000; Böhm et al. 2011; Bruening et al. 2003; Chen et al. 2000; D'Antras et al. 2017; Gal et al. 2009; Hong et al. 2012; Inoue et al. 2011; Wang et al. 2007; Wu et al. 2011) have adopted NET or its variants to conduct trace formation. However, the NET selection algorithm faces the problems of trace separation and early exits. For addressing these issues, numerous methods have been proposed in the literature.

Trace formation methods. MRET2 (Two-pass Most Recent Execution Tail) (Wang et al. 2010) is a two-pass trace selection algorithm. It performs trace selection twice from the same trace head, and the final hot trace is the common path of the two selected traces. Using this two-pass selection, MRET2 reduces the possibility of selecting a bad trace that causes frequent trace exits. Hayashizaki et al. (2011) found that a function is falsely identified as a loop by the trace formation algorithm if the function is invoked multiple times in a loop body, which results in the real loop being split into short traces. They proposed a false-loop filtering mechanism, which monitors the call stack. Trace selection continues through the function until the real loop is formed in one trace. Inspired by the work of Hayashizaki et al. (2011), Castanos et al. (2014) proposed the N-E-C (Next-Executing-Cycle) method, which is an adaptive trace selection algorithm. N-E-C defines two-level thresholds. When the trace head reaches the level-1 (i.e., lower) threshold, a hot trace is selected using the NET algorithm, which terminates trace recording if an existing trace head is encountered. When the trace head reaches level-2 (i.e., higher) threshold, the trace selection continues through an existing trace head instead of being terminated. Thus, a long trace, which can select the loop mentioned in Hayashizaki et al. (2011), is formed. D'Antras et al. (2017) addressed the issue that hardware return address prediction cannot be exploited if the formed traces do not preserve the call/return structure. To overcome this problem, they proposed an improved NET algorithm that stops trace recording when the call/return sequence is detected. The aforementioned algorithms use different termination conditions to lengthen traces, avoid early exits, or exploit hardware features. However, these algorithms still focus on forming traces, which do not solve the problems of trace separation and early exits for complex regions.

Hiniker et al. (2005) proposed the Last Executed Iteration (LEI) trace formation algorithm. In contrast to the NET algorithm that selects traces from the next executed blocks, LEI selects cyclic

traces from a software-implemented history buffer containing the most recently taken branches. Although the LEI algorithm achieves less trace separation and code duplication than NET, its implementation introduces high profiling overhead. ADORE (Lu et al. 2004) is a lightweight dynamic optimization system based on sampling-based processor tracing. It collects path profiles from the Itanium II BTB counters, and the four most recently taken branches are recorded at each interrupt. When a set of four branches occurs frequently, the corresponding path is selected and linked with other frequent paths to form a trace. The LEI algorithm and ADORE system do not form regions.

Region formation methods. The IA-32 EL software (Baraz et al. 2003) applies two-phase translation. The first phase instruments cold blocks to collect the block and edge frequencies, which are used for hot trace selection in the second phase. A trace is further extended to a region containing if-then or if-then-else structures if predication can be used to include these structures as part of the linear trace. Unlike IA-32 EL, our method does not employ instrumentation and builds general regions of any shape. The formation of general regions has been reported in the following DBT systems: the EHS simulator (Jones and Topham 2009), Transmeta CMS (Dehnert et al. 2003), and the x86 simulator (Borin et al. 2010). The EHS simulator identifies hot regions by profiling block execution using interpretation, whereas our algorithm collects execution profiles through processor tracing. Transmeta CMS requires special hardware to form regions, however, its design details are not reported in the literature. Borin et al. (2010) presented the frame formation logic, which is a new hardware component that can profile retired micro-operations and use branch predictability information to assist trace and region formation. The functionality of the frame formation logic is similar to processor tracing. However, their work is evaluated on the simulator level, whereas our method is designed with real hardware.

Numerous extensions of the NET selection algorithm have been proposed for region formation. The NETPlus algorithm (Davis and Hazelwood 2011) forms regions by combining a sequence of basic blocks that can flow back to the NET trace head. We demonstrated in Section 4.4 that our algorithm achieves better performance and lower compilation overhead compared with NETPlus. Hong et al. (2012) and Hsu et al. (2013) proposed the trace merging algorithms based on HPM sampling. Similarly to NETPlus, they begin with NET trace formation but no additional block is combined in the first phase. Instead, HPM sampling is used to monitor the execution of NET traces. In Hong et al. (2012), multiple connectable traces are merged into one region if the traces are frequently sampled. In Hsu et al. (2013), the exit points of frequently sampled traces are instrumented with profiling counters. Two traces are combined if the associated profiling counter reaches a certain threshold. These algorithms perform NET trace formation in the first phase that involves instrumentation, and additional paths are included in the second phase to form regions. In contrast, our algorithm is a non-intrusive approach. Regions are formed by utilizing branch records collected through processor tracing.

Sampling methods. Sampling is a common mechanism to reduce the overhead of profiling: Instead of exhaustively collecting information, profiling is only applied at specific time points. Sampling can be driven by software or hardware. Software sampling typically uses operating system features or software-based counters as the trigger mechanism. Jikes RVM (Alpern et al. 2000) and IBM J9 VM (Sundaresan et al. 2006) use the OS timer to trigger a sampling thread, whereas the OS sleep function is used in Whaley (2000) to periodically awakes a profiler thread to sample the call stacks of application threads. Using counter-based sampling, Arnold and Ryder (2001) proposed a framework that duplicates execution code in two versions: One version is instrumented to collect profile information and the other version is only inserted with checking counters in procedure entries and loop backedges. Their framework samples by using the checking counters and switches the execution to instrumentation code only when the counter value decreases to zero. Hirzel and Chilimbi (2001) further enhanced the Arnold-Ryder framework. The enhancement allows profiling

to span across procedure boundaries, and overhead is reduced by executing fewer counter checks. Unlike the aforementioned software-based sampling approaches, our method uses HPM sampling. Tam and Wu (2003) and Buytaert et al. (2007) are two related works that also leverage HPM sampling to identify hot methods for optimization. Tam and Wu (2003) instrument method prologues and epilogues to determine the hotness of methods by reading the HPM cycle counter; Buytaert et al. (2007) find optimization candidates by retrieving method ID from the stack on the HPM counter overflow. These work aim to dynamically recompile code for higher optimization levels. In contrast, our work uses HPM sampling for triggering hot region formation and minimizing the overhead of processor tracing.

6 CONCLUSION

In this article, we present a processor-tracing guided region formation method. We leverage the branch history recorded by the processor and construct an execution flow graph containing node and edge frequencies. According to the reaching probabilities derived from the node and edge frequencies, general regions are formed by selecting only the basic blocks with high reaching probabilities. Because of this probability-based region formation, the execution is likely to remain in the formed regions, resulting in minimal region exits. We implemented our method in a retargetable cross-ISA DBT, which is based on the popular NET trace formation method. For the ARM64 to x86-64 translations with the SPEC CPU2006 benchmarks, our method outperforms NET with a speedup of up to 1.53× (1.16× on average). To minimize the system overhead, we design a mechanism that combines processor tracing and lightweight HPM sampling. This mechanism effectively limits the region formation overhead to less than 8%. Moreover, the design of the branch instruction decode cache further reduces the overhead to less than 3%. The comparison with a relaxed NETPlus region formation algorithm demonstrates that our method achieves the best performance and lowest compilation overhead.

ACKNOWLEDGMENTS

The authors thank the reviewers for their valuable comments and suggestions to improve the quality of this article.

REFERENCES

B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. 2000. The jalapeñO virtual machine. *IBM Syst. J.* 39, 1 (Jan. 2000), 211–238.

ARM. 2012. *CoreSight Components Technical Reference Manual.* ARM.

Matthew Arnold and Barbara G. Ryder. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation.* 168–179.

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* 1–12.

Thomas Ball and James R. Larus. 1994. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.* 16, 4 (Jul. 1994), 1319–1360.

Thomas Ball and James R. Larus. 1996. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture.* 46–57.

Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. 2003. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture.*

Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference.* 41–46.

Igor Böhm, Tobias J. K. Edler von Koch, Stephen C. Kyle, Björn Franke, and Nigel Topham. 2011. Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation.* 74–85.

Edson Borin, Youfeng Wu, Cheng Wang, Wei Liu, Mauricio Breternitz, Jr., Shiliang Hu, Esfir Natanzon, Shai Rotem, and Roni Rosner. 2010. TAO: Two-level atomicity for dynamic binary optimizations. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 12–21.

Derek Bruening, Timothy Garnett, and Saman Amarasinghe. 2003. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*. 265–275.

Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. 2007. Using Hpm-sampling to drive dynamic compilation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*. 553–568.

J. G. Castanos, H. Hayashizaki, H. Inoue, M. J. Serrano, and P. Wu. 2014. Adaptive next-executing-cycle trace selection for trace-driven code optimizers. http://www.google.com/patents/US8756581 US Patent 8,756,581.

Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. 2000. Mojo: A dynamic optimization system. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*. 81–90.

Amanieu D'Antras, Cosmin Gorgovan, Jim Garside, and Mikel Luján. 2017. Low overhead dynamic binary translation on ARM. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 333–346.

Derek M. Davis and Kim Hazelwood. 2011. Improving region selection through loop completion. In *Proceedings of the ASPLOS Workshop on Runtime Environments/Systems, Layering, and Virtualized Environments*.

James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. 2003. The transmeta code morphing™ software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. 15–24.

Evelyn Duesterwald and Vasanth Bala. 2000. Software profiling for hot path prediction: Less is more. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*. 202–211.

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*.

Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*. 120–126.

Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio J. Serrano, and Toshio Nakatani. 2011. Improving the performance of trace-based systems by false loop filtering. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 405–418.

David Hiniker, Kim Hazelwood, and Michael D. Smith. 2005. Improving region selection in dynamic optimization systems. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. 141–154.

Martin Hirzel and Trishul Chilimbi. 2001. Bursty tracing: A framework for low-overhead temporal profiling. In *Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization*.

Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Yeh-Ching Chung, Pangfeng Liu, and Chien-Min Wang. 2012. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the International Symposium on Code Generation and Optimization*. 104–113.

Chun-Chen Hsu, Pangfeng Liu, Jan-Jan Wu, Pen-Chung Yew, Ding-Yong Hong, Wei-Chung Hsu, and Chien-Min Wang. 2013. Improving dynamic binary optimization through early-exit guided code region formation. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 23–32.

Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. 2011. A trace-based Java JIT compiler retrofitted from a method-based compiler. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. 246–256.

Intel Corporation 2018. *Intel(R) 64 and IA-32 Architectures Software Developer's Manual: Volume 3*. Intel Corporation.

Daniel Jones and Nigel Topham. 2009. High speed CPU simulation using LTU dynamic binary translation. In *Proceedings of the 4th International Conference on High Performance Embedded Architectures and Compilers*. 50–64.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*. 75–88.

Linaro. 2018. OpenCSD library. Retrieved from https://github.com/Linaro/OpenCSD.

Linaro ToolChain. 2017. Linaro ARM GCC toolchain. Retrieved from http://www.linaro.org/downloads/.

Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. 2004. Design and implementation of a lightweight dynamic optimization system. *J. Instruct.-Level Parall.* 6 (2004), 1–24.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 89–100.

Andreas Neustifter. 2010. *Efficient Profiling in the LLVM Compiler*. Master's thesis. Vienna University of Technology.

Vijay Sundaresan, Daryl Maier, Pramod Ramarao, and Mark Stoodley. 2006. Experiences with multi-threading and dynamic class loading in a java just-in-time compiler. In *Proceedings of the International Symposium on Code Generation and Optimization*. 87–97.

David Tam and John Wu. 2003. *Using Hardware Counters to Improve Dynamic Compilation*. Technical Report.

Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2002. Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. 86–96.

Cheng Wang, Shiliang Hu, Ho-seop Kim, Sreekumar R. Nair, Mauricio Breternitz, Zhiwei Ying, and Youfeng Wu. 2007. StarDBT: An efficient multi-platform dynamic binary translation system. In *Proceedings of the Asia-Pacific Conference on Advances in Computer Systems Architecture*. 4–15.

C. Wang, B. Zheng, H. S. Kim, M. Breternitz, and Y. Wu. 2010. Two-pass MRET trace selection for dynamic optimization. http://www.google.com/patents/US7694281 US Patent 7,694,281.

John Whaley. 2000. A portable sampling-based profiler for java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande*. 78–87.

Peng Wu, Hiroshige Hayashizaki, Hiroshi Inoue, and Toshio Nakatani. 2011. Reducing trace selection footprint for large-scale java applications without performance loss. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. 789–804.

Received June 2018; revised August 2018; accepted September 2018