Improving Startup Performance in Dynamic Binary Translators

Surya Tej Nimmakayala
EECS, University of Kansas
Lawrence, KS USA
s387n230@ku.edu

Prasad A. Kulkarni
EECS, University of Kansas
Lawrence, KS USA
prasadk@ku.edu

***Abstract*—A Dynamic Binary Translation (DBT) system dynamically translates program binaries built for a *guest* platform into code for the *host* machine that runs the program, one basic block at a time. Even after optimizations, auxiliary tasks performed alongside program emulation by the DBT system introduce performance overheads as compared to executing the program on the native guest platform. In this work, we analyze the extent and causes for a DBT system's startup performance latency. We then focus on understanding and alleviating the *program translation* cost that is a significant contributor to and disproportionately impacts the startup overhead. We propose and assess the potential of a new technique that parallelizes program translations on multi-core machines to reduce its evident run-time costs. We explain the challenges in achieving such parallelization and discuss and evaluate solutions.**

***Index Terms*—Dynamic binary translation, performance**

I. Introduction

Dynamic binary translation (DBT) systems take a *guest* binary program as input and dynamically translate it at run-time to the *host* machine for execution. The guest and host platforms may or may not employ the same instruction set architecture (ISA) and/or operating system (OS). Dynamic binary translation is a key technology that enables portable binary execution [1]–[7], system virtualization [8], program debugging [9], program instrumentation [10]–[12], dynamic optimizations [13]–[15], and code analysis and transformations for secure execution [16], [17].

Execution of the guest binary on the host platform requires a DBT system to perform several auxiliary tasks, including program translation and resolution of indirect control transfers, in addition to emulation. Many of these auxiliary tasks are conducted inline and stall the guest program emulation. The resulting run-time costs lower the DBT system's performance on the host system as compared to native execution of the program on an equivalent guest platform.

The performance of a DBT-like runtime system is commonly studied and evaluated as two related, but distinct issues [18]: (a) program *startup* performance that measures the throughput for short-running programs and latency for applications that interact with or provide feedback to users, (b) program *steady-state* performance that measures the throughput for longer-running applications. Even as sophisticated optimizations employed by current DBT systems have diminished the steady-state performance overhead, startup performance still remains significant for most systems.

The goal of this work is to study the main causes of performance overhead during program startup and suggest and evaluate techniques to overcome major issues. We find that the block translation overhead along with associated costs, such as context switching, are some of the major factors that disproportionately impact a DBT system's startup performance, compared to steady-state performance later in the run.

The second part of this paper explores the problem of reducing the translation overhead at program startup. On multi/many-core machines, one obvious technique to reduce the user-evident block translation overhead is to translate blocks in parallel on free processor cores. We demonstrate the potential and challenges of this approach with detailed experiments and two techniques, one *offline* and the other *online*. We discuss our observations and remaining challenges.

This work makes the following contributions.

- We highlight the problem of program startup performance and study causes of startup overhead in DBT systems.
- We study how various auxiliary tasks, especially the translation overhead, differently impact program startup versus steady-state performance for DBT systems.
- We propose and evaluate a novel technique to reduce translation overhead on multicore machines.
- We discover the remaining challenges in satisfactorily resolving the translation overhead in DBT systems.

II. Background

In this section we provide some background on dynamic binary translation systems that is relevant to this work. DBT systems use a technique called *incremental translation* to *discover* and translate the guest code one *basic block* at a time as they are reached during execution. The translated blocks are placed in a region of heap memory, called the *code cache* for later reuse. As execution reaches the end of each translated block, control is returned back to the DBT engine (called a *context switch*), which determines the next block to execute. The DBT engine checks if this next block is already in the code cache, and if not, employs the compiler to translate the next block. Additionally, the DBT engine generally maintains a guest PC to target PC map table and dynamically performs the mapping to resolve targets of indirect branches at run-time.

Thus, in addition to program emulation, the DBT system performs several auxiliary tasks during execution, including code translations, indirect branch resolution and system call handling. Most auxiliary task events occur along the critical path of program emulation and stall the program execution until that event is resolved. Additionally, each auxiliary task typically requires a context switch from the code cache to the DBT engine, and back. The DBT system may need to save and restore program state at context switches. These context switches and inline execution of auxiliary tasks produce performance overheads in DBT systems.

III. Related Works

Researchers have proposed, built, and evaluated several techniques to reduce the performance impact of auxiliary tasks for DBT systems. Optimization techniques, such as introducing a basic block *code cache* and block *chaining*, help both startup and steady-state performance. Other optimizations, such as *trace generation*, are designed to improve performance for long-running programs. In this section we present prior research in areas of DBT optimization techniques targeted to improve warmup or startup DBT performance.

In the past, several researchers have studied the performance impact of different aspects of DBT. Ruiz-Alvarez and Hazelwood analyzed the effect of the changed application code layout in a DBT system on microarchitectural performance [19]. This study found that the increase in the number of instructions executed (compared to native execution) due to DBT interference is the main factor affecting program performance. This study analyzed performance for long-running programs. In contrast, our work explores how auxiliary tasks affect program performance during startup execution.

Several researchers have studied persistent code caching to reduce startup overhead in DBT systems [20]–[22]. Persistent code caches enable code reuse by storing and reusing translations across executions, and techniques even exist to handle dynamically generated code [21]. These approaches can achieve high performance during startup by reducing the translation overhead. However, the persistent code cache itself needs to be warmed up, is not as effective when an unseen program without representation in the code cache is executed, and raises some security concerns [23].

Multi-level compilation is a model for DBTs [24] and other runtime systems [25] to improve program startup performance without compromising on good translated code quality that is necessary for high performance on long-running programs. A multi-level compilation scheme provides fast code translation initially, with the important code regions later recompiled with slower high-quality translators. These slower compilers are typically designed to operate as separate parallel threads. However, for systems without an interpreter, the first mandatory translation still blocks the application thread. Our work attempts to parallelize all translation activity and is complimentary to multi-level compilation.

IV. Experimental Configuration

We use the *DynamoRIO* DBT system for this study [11], [26]. DynamoRIO is an open-source runtime code manipulation system that employs dynamic binary translation to support code transformations on the guest program, while it executes. DynamoRIO only allows configurations with identical guest and host systems. Translations in DynamoRIO are typically very quick since they maintain identical guest code to the host code mapping (x86-64 for this study), unless instrumentations or other changes are explicitly requested by the user. This methodology contrasts with DBT systems designed for portability, like Qemu [1], [27], which maps guest instructions to an intermediate representation before translating to a potentially different host machine format. Thus, DynamoRIO offers a DBT system with a very low translation overhead.

We employ the SPEC cpu2006 benchmark suite, which provides industry standard, CPU-intensive single-threaded benchmarks [28]. We discard a few benchmarks that fail on some of our DynamoRIO configurations. Each SPEC cpu2006 benchmark provides three input workloads, *test*, *train*, and *ref*. In this study we employ the short-running *test* workloads to deliver measurements pertaining to benchmark startup performance, and the longer-running *ref* workloads to furnish steady-state performance measurements.

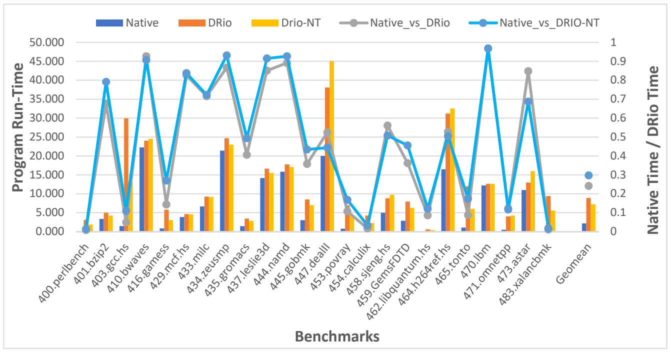
Our experiments were conducted on a cluster of identically configured Intel Xeon (R) E5430 2.66 GHz (x8 cores) workstations with 16GB memory. All machines run the 64-bit CentOS Linux release 7.2.1511 operating system. All benchmarks are compiled using GCC version 4.8 and optimized with the *-O2* flag. All experiments run each configuration 10 times, and the average measurement is plotted or reported.

V. Analyzing DBT Behavior at Program Startup

We have two primary goals with this work for DBT systems: (a) to explore the magnitude of performance overhead and the contribution of the translation cost at program startup compared to steady-state, and (b) to develop techniques to reduce translation overhead at program startup. In this section we examine and contrast performance and translation behavior at program startup versus program steady-state.

a) DynamoRIO versus Native Performance: Our first experiment compares DBT startup and steady-state performance with *native* program performance. With DynamoRIO, the input guest code and generated host code use the same x86-64-Linux machine configuration. The first bars for each benchmark in Figures 1(a) and 1(b) plot the native benchmark execution time when using the *test* and *ref* input data sets, respectively. With native execution, programs run for 2.1 seconds and 474.6 seconds with the *test* inputs and *ref* inputs, respectively. The next benchmark bars display the program execution time when running with the default DynamoRIO DBT system. We find that running with DynamoRIO incurs a performance penalty of 75.9% and 17.4% with the *test* and *ref* inputs, respectively.

The final bars for each benchmark in Figures 1(a) and 1(b) show program run-time with DynamoRIO configured with *traces* disabled. A *trace* is a sequence of frequently executed


(a)

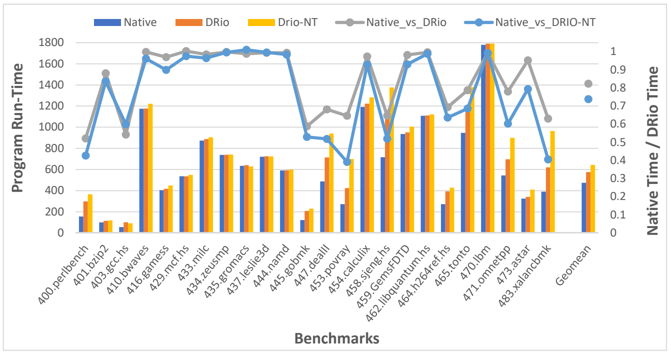

(b)

Fig. 1. DynamoRIO emulation time compared to the native execution time for each benchmark on the same platform, for the (a) test and (b) reference SPEC inputs, respectively. The bars are plotted on the left Y-axis (lower is faster), while the line-plots use the right Y-axis (higher is better).

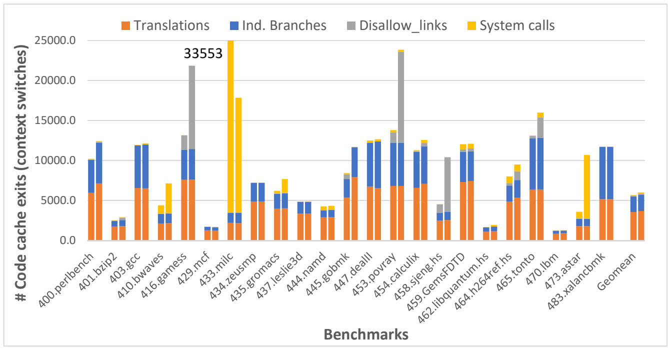


Fig. 2. Number of DynamoRIO code cache exits with the *test* (first benchmark bar) and *ref* (right benchmark bar) inputs. Although programs with *test* inputs run for much shorter duration as compared with the *ref* inputs, they encounter a similar number of total code cache exits during program execution.

basic blocks placed together as one unit. Trace construction is an optimization that can improve the efficiency of indirect branches and achieve better code layout [11]. Without trace construction, benchmark performance with DynamoRIO degrades about 10% for *ref* inputs, but improves by 19% for *test* inputs. Given our focus on improving startup program performance, and to simplify DynamoRIO code modifications, all our later experiments in this study disable traces.

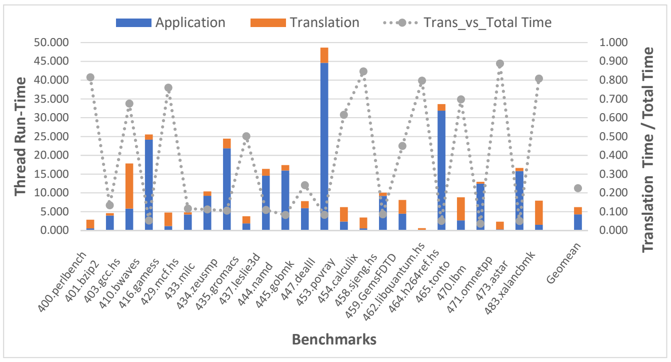
b) Context Switch: A typical DBT system translates the guest application code one block at a time and stores it in the code cache. This cached code can then be natively executed. The DBT system enters/leaves the code cache via a mechanism known as a (user-level) *context switch*. Occasionally, the execution needs to switch back to the DBT emulator to perform auxiliary tasks, including translating guest code that is not already in the code cache, resolution of indirect control transfers, and system call processing. Context switches require the DBT system to save and restore program state like general-purpose registers, condition codes and other operating-system dependent state. Thus, context switches are expensive, and DBT systems, including DynamoRIO, often perform several optimizations, such as branch chaining and inline caching of indirect branch/call targets, to lower their occurrence.

Figure 2 plots the number of code cache exits for the DynamoRIO DBT with the *test* and *ref* inputs. The context-switches are categorized according to their purpose into, (a) translation, (b) indirect branch resolution, (c) block linking disallowed, and (d) system call handling. It is not surprising to find that block translation and indirect branch resolution are responsible for most context switches from the code cache. However, it is remarkable that although the average program run-time with the *ref* input increases by a factor of 72X over the *test* input, the average number of context switches only increases by a factor of 1.1X. This relatively modest increase is because the DBT system uses the program startup period to perform block translations and other optimizations that reduce context switches during later execution. Thus, the context switch overhead largely impacts program startup or early program execution performance.

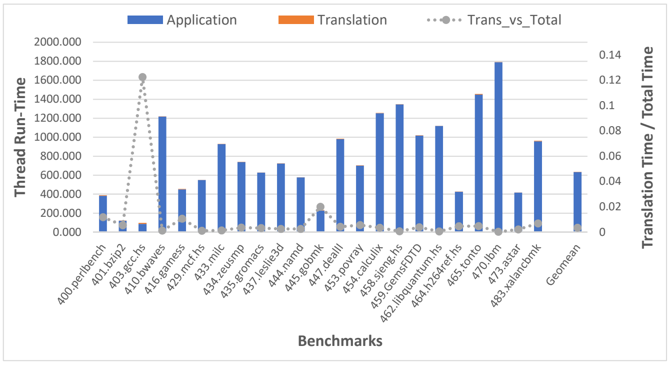
c) Translation versus Application Time: By default, DynamoRIO conducts block (and trace) translation synchronously with application execution in the *same* thread. We constructed a new DynamoRIO configuration with *separate* application and translation threads. While translation still occurs in lock-step with program emulation for the experiments in this section, this new DynamoRIO configuration could be used to perform code translations *asynchronously* with and in parallel with program execution. A global FIFO queue serves as the interface between the application and compiler thread(s).

Our next experiment is designed to measure the translation overhead in DBT systems during program startup and steady-state. We use our new DynamoRIO configuration (with a single compiler thread) for this experiment. The application thread(s) make a translation request the first time a code block is reached during execution and/or the block is not present in the code cache. Such requests stall the application thread until the translation is completed. We call them the *urgent* translation requests. We use thread execution times given by Linux's *proc* file-system [29] to report application and translation thread times.

The stacked bars in Figures 3(a) and 3(b) plotted on the left Y-axis show the actual application and translation times



(a) Test Input



(b) Ref Input

Fig. 3. Block translation time compared to all other time spent during program execution with DynamoRIO for the (a) test and (b) ref SPEC inputs, respectively. The stacked bars are plotted on the left Y-axis, while the line-plot uses the right Y-axis. Translation time is a significant component of total program execution time for shorter running programs (with a DBT system).

for each benchmark with the *test* and *ref* inputs respectively. The line graph plots the ratio of the benchmark translation time and application run-time on the right Y-axis. While block translation overhead only accounts for 0.3% of total program execution time with the *ref* inputs, it comprises a much more significant 22.5% (and as high as 90%) with the *test* inputs, on average. Therefore, translation cost is a major factor for performance overhead at program startup. In the next section we propose and evaluate a new technique to reduce the evident block translation and context switch costs of a DBT system on multi-core machines.

VI. Parallelizing Block Translations

Many Java and other high-level language runtime systems, such as the HotSpot Java virtual machine [25], include an interpreter in addition to the just-in-time (JIT) compiler to perform emulation of the guest binary. In such systems the JIT compilations facilitate performance enhancement over interpreted execution, but do not block the execution of the application. These systems reduce compilation activity by supporting *selective compilation*, where program units (traces or methods) are only compiled/optimized if they are frequently executed (or *hot*). Selective compilation not only reduces the compilation activity, but also allows the important program units to be compiled sooner. Selected program units are queued by the application thread(s) as and when they are detected. Separate compiler threads work asynchronously to generate optimized native code. Even with selective compilation, JIT compilers in a JVM notice a surge in compilation activity at program startup. Many such systems spawn multiple compiler threads to more effectively handle this surge and to allow programs earlier access to native code. Native code execution provides the runtime with a higher performance emulation alternative over interpretation.

Our proposed block translation parallelization framework in DBT systems is inspired by such language runtime systems. However, the two systems have important differences. DBT systems, like DynamoRIO, Pin and QEMU, lack an interpreter. Therefore, all block translation requests in such DBT systems are blocking (or *urgent*). Since application threads stall on every translation request sent, there is at most one pending request from each application thread at any given time. There is no provision to locate, queue and translate blocks ahead-of-time. Additionally, since all code units reached during execution first require guest-to-host translation, there is typically more translation activity in terms of number of requests as compared to a runtime with selective compilation.

Therefore, our first challenge to parallelize block translations is to find guest binary code blocks before they are reached during execution. We call this speculative translation of blocks as *eager* translation. We examine and present two eager translation techniques. The first technique uses an *offline* approach to generate a list of block addresses that are guaranteed to be needed during program execution (Section VI-B). This method provides an ideal baseline to evaluate the potential benefit of eager translation. Our next technique generates the list of eager block addresses *online* (Section VI-C). First, we describe the experimental framework we employ for this study in the next section.

A. Framework to Study Parallelizing Block Translations

As described earlier in Section V-c, we extended the DynamoRIO framework to asynchronously isolate the translation activity in separate threads and provide a request queue for application thread(s) to interface and communicate with the translation thread(s). This setup allows us to create multiple translation threads. However, DynamoRIO's emulation and translation frameworks were not designed to operate in parallel and make heavy use of shared global structures and locks. We found that DynamoRIO's current design prevents the translation threads from actually operating concurrently with each other or with the application threads.

Our focus for this work is to evaluate the potential of parallel translations on DBT startup performance and identifying other challenges to achieve this potential. While we realize that issues involving shared data structures, locks, race conditions, etc. can reduce the ideal benefits from parallelization and that resolving DynamoRIO's design issue is a substantial research

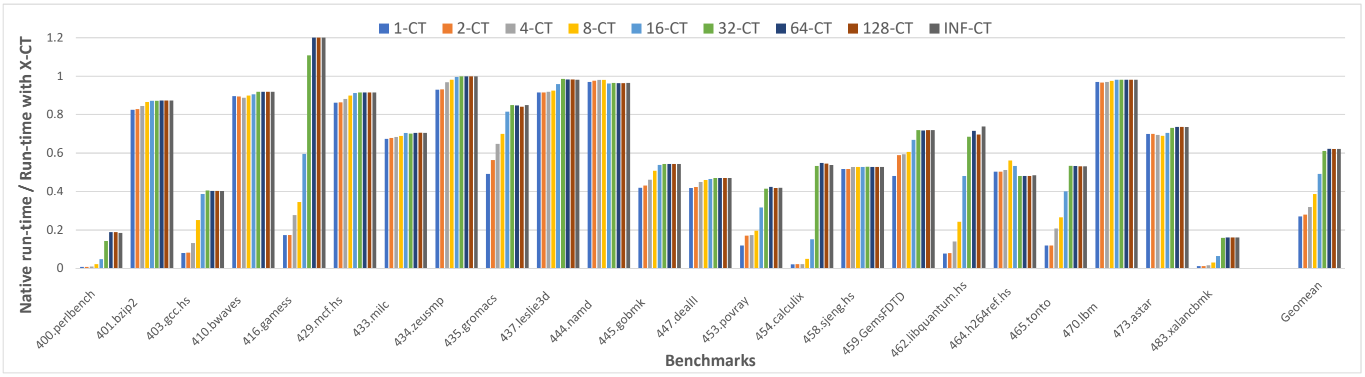


Fig. 4. Application performance with a DBT (compared to native program run-time) that uses varying number of parallel translation threads using the *offline* approach to generate block addresses to eagerly translate. Performance improves rapidly with a small number of parallel translation threads.

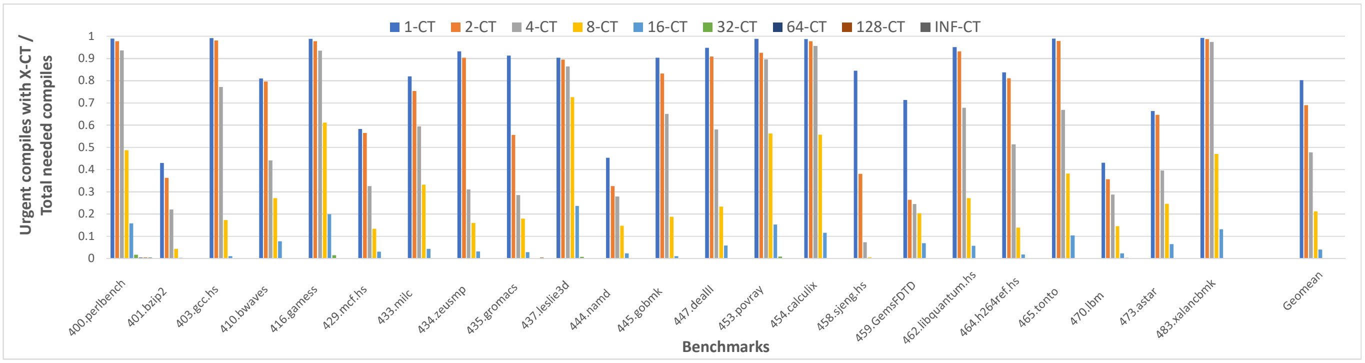


Fig. 5. Ratio of necessary block compiles that are *urgently* compiled with a DBT configuration that uses varying number of parallel translation threads using the *offline* approach to generate block addresses to eagerly translate. A small number of parallel translation threads are able to significantly decrease the number of (urgent) translations that stall application progress.

and engineering challenge, we decided to leave a significant architectural overhaul of the DynamoRIO DBT system to future work. (We do note that parallel JIT compilers have been successfully built for several runtimes [25].)

Instead, for this work we build a simple and precise framework to emulate the behavior of multiple parallel translation threads with a single thread. Our translation thread operates in lock-step with the application thread(s) on a single processor core. The translation thread is invoked on an urgent request from an application thread. On entry, the translation thread extracts the total application thread time and multiplies this time by 'N', where 'N' is the total number of translation threads being emulated. This process gives us the time available to the translation thread to do its work. The translation thread locks the processor until this calculated time is exhausted or if there is nothing left to translate in the queue. The time taken to process all the urgent requests that invoke the translation thread is stored (lets call it the *urgentReqProcTime*), since the application threads are stalled for urgent requests. Finally, on program termination, our DBT system fetches the thread times from the *proc* file-system. The application thread times combined with the *urgentReqProcTime* gives us the total program run-time. All other compilation activity is reported as having occurred in parallel with actual program execution.

B. *Offline Approach to Generate Eager Block Addresses*

In this section we present our offline profiling based approach to generate a list of guest binary block addresses for the parallel compiler threads to speculatively translate. This approach allows us to determine the performance potential of eagerly translating blocks in DBTs to minimize evident translation overhead.

The offline approach works in two steps. First, we run each benchmark program (with the *test* input data-set) with a DynamoRIO configuration that outputs the list of block addresses that are reached during execution. The later measured runs employ a DynamoRIO setup that uses this list of block addresses to populate the compiler queue during initialization. This setup then employs the procedure described in the last section that uses the addresses in the compiler queue to guide the eager translation of blocks in parallel with application execution. The measured runs also use the *test* input, so they only translate blocks that are guaranteed to be reached during the program execution. We disable the OS' ASLR (Address space layout randomization) facility for this experiment.

This experiment launches measured runs with the DBT system simulating varying number of translation threads. Figure 4 shows the application performance with 1, 2, 4, 8, 16, 32, 64, 128, and INF (unlimited) parallel translation threads. The

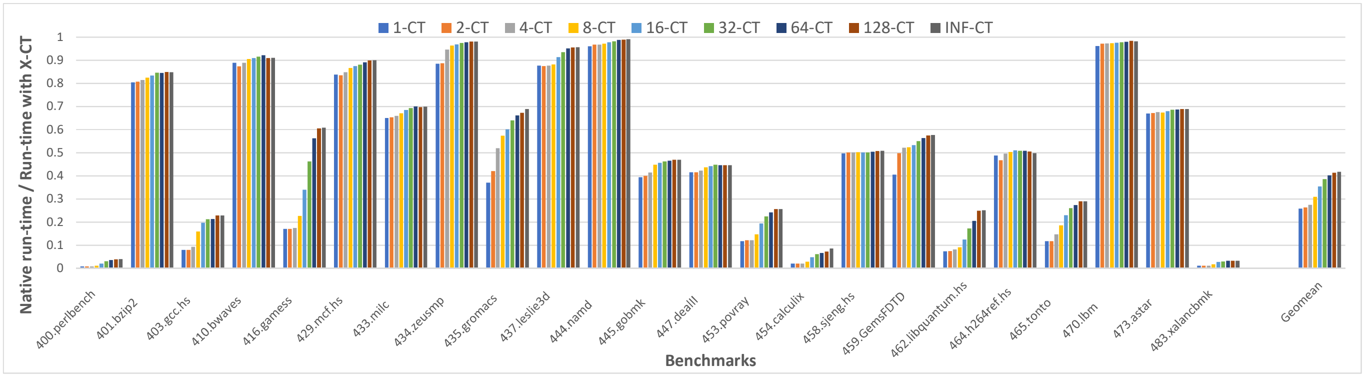


Fig. 6. Application performance with a DBT (compared to native program run-time) that uses varying number of parallel translation threads using the *online* approach to generate block addresses to eagerly translate. Performance improvement with a small number of parallel translation threads is significant, but at 42% of native performance, much lower than the ideal *offline* approach.

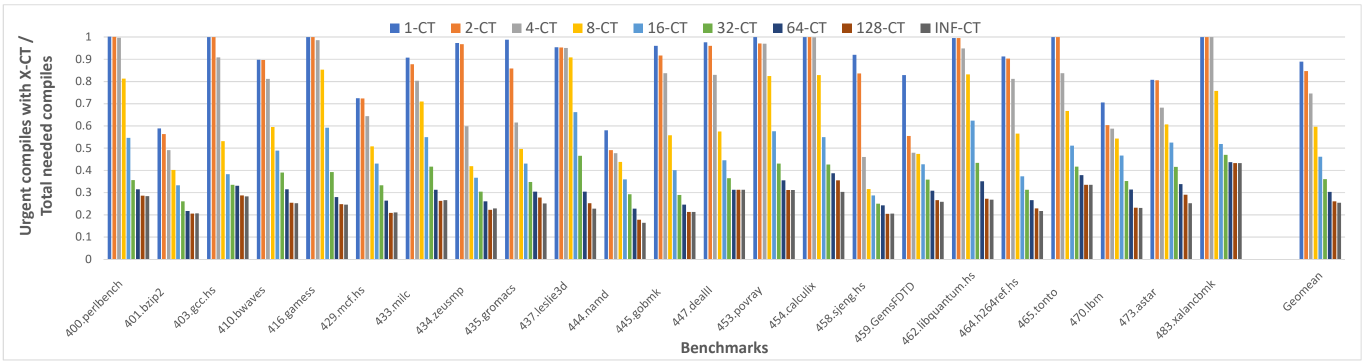


Fig. 7. Ratio of necessary block compiles that are *urgently* compiled with a DBT configuration that uses varying number of parallel translation threads using the *online* approach to generate block addresses to eagerly translate. Urgent translation requests reduce by as much as 74%, on average.

application performance is displayed by comparing its run-time (derived using the procedure described in Section VI-A) with native guest program run-time on the same platform. We see that performance improves rapidly with increasing number of parallel translation threads. The performance benefit is due to the combined effect of fewer urgent compiles blocking application progress and fewer DBT context switches. We find that a small number of compiler threads is sufficient to get most of the performance benefits. Overall, performance improves from 27% to 62% of native program execution-time with 1 to 128 parallel translation threads, on average[1].

Figure 5 plots the ratio of the number of urgent translations performed with this DBT configuration and varying number of parallel translation threads divided by the total number of necessary block translations for each benchmark. Eager translations with this DBT configuration always successfully reduce urgent translation requests. We find that even a single translation thread with this setup reduces the number of urgent translation requests by 20%, on average. Ideally, the *INF-CT* configuration should result in only a single urgent request for each run. However, the offline profiling run is not always able to accurately find all block addresses for a few benchmarks. This limitation causes the DBT to make about 2.9 urgent translation requests with the *INF-CT* configuration, on average. Overall, we conclude that accurate speculative/eager translation with a small number of parallel translation threads can successfully raise DBT startup performance significantly.

[1]Performance of the 416.gamess benchmark exceeds its native execution time, which may be due to a better block layout and caching effects that we are yet to explore more fully.

C. *Online Approach to Generate Eager Block Addresses*

In this section we describe and evaluate a simple online approach to automatically generate the list of block addresses to eagerly translate during program execution. This online approach uses a greedy algorithm to locate valid block start addresses and translate the maximum number of blocks before the application terminates. For every translated block that ends in a conditional branch, this online technique adds two addresses, the fall-through and target, to the queue that provides the work-list for the translation threads. For blocks containing direct jump and call instructions, DynamoRIO already continues translation from the target location and includes those instructions in the same block. We do not yet add any addresses for blocks ending with indirect control transfers (indirect jumps, branches, calls, or returns).

Other than updating how the speculative block addresses are generated, the DynamoRIO framework and experimental

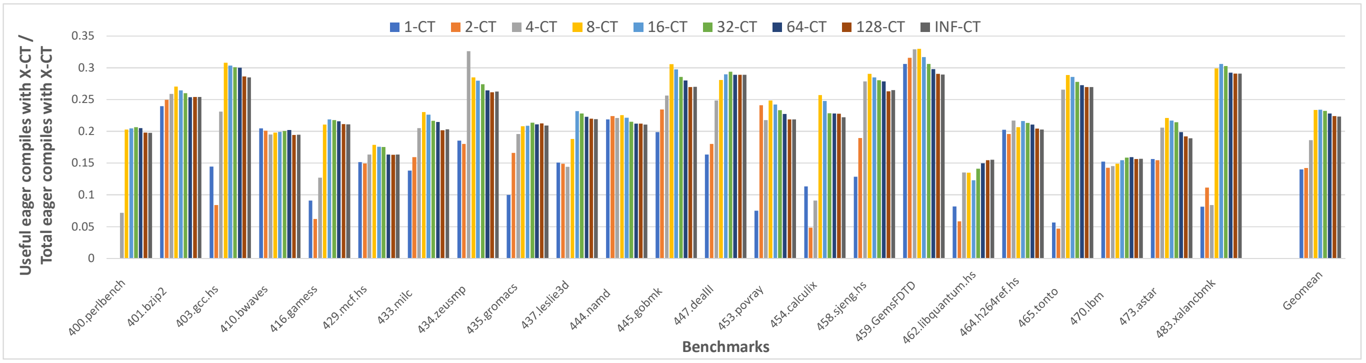


Fig. 8. Ratio of eager translations performed by our *online* approach that are used by the application thread. Results show a low rate of successful predictions.

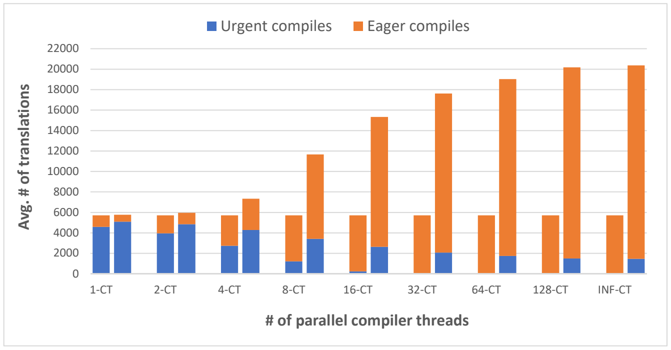


Fig. 9. Average number of urgent and eager translations performed for each benchmark by *offline* (first bar) and *online* (second bar) approaches.

setup used in this section are identical to that employed in Section VI-B. Figure 6 compares the application performance with 1, 2, 4, 8, 16, 32, 64, 128, and INF (unlimited) parallel translation threads with native program run-time, and plots their ratio. Comparing these results with the ideal baseline from Figure 4 we can see that while performance improves noticeably with increasing number of parallel translation threads, the improvement significantly lags the ideal baseline. Overall, performance improves from 26% of native program execution-time to 42% of native execution-time with 1 to 128 parallel translation threads, on average.

Similar to Figure 5, Figure 7 plots the ratio of the number of urgent translations performed with this DBT configuration and varying number of parallel translation threads divided by the total number of necessary block translations that are needed for each benchmark. On average, the number of urgent compiles reduce by 11% with a single translation thread, and up to 74% with 128 translation threads, compared to a system with no eager translations. Interestingly, while this online eager translation configuration is able to eliminate about 75% of the urgent requests in the best case, the best performance benefit derived (at 42% of native performance) is much lower than 75% (which would be, $26 + 0.75 * (62 - 26) = 53\%$), of the baseline benefit with the offline approach. We study the efficiency of the online approach to explore this observation.

Figure 8 plots the ratio of *useful* eager translations, that is correct speculative translations that successfully eliminate needed urgent translation requests. This graph indicates that our simple online approach is quite inefficient in finding useful blocks to translate. In the best case, only 23% of the eagerly translated blocks are useful, with this number remaining relatively stable after about 16 translation threads.

Figure 9 displays the total number of urgent and eager translations performed with the offline and online approaches for varying number of translation threads, averaged over all the benchmarks. The offline approach only translates known useful blocks, and we can see that almost all blocks are eagerly translated after about 16 translation threads. In contrast, the online approach is not able to locate a significant number of block addresses that are reached during execution, even in the INF-CT configuration. We also find that the online approach with multiple translation threads processes a very large number of blocks that are then placed in the code cache, increasing the DBT's memory usage. Poor locality in the code cache and large memory usage impacting processor cache performance may be the reason for low performance numbers with the online technique. A secondary, but nonetheless interesting observation is that large parts of the program seem to not be reached during execution. We require further research to develop algorithms and techniques that improve the efficiency of useful eager translations.

VII. Future Work

Future directions for this work span three goals. First, develop new techniques and algorithms to improve the predictability of eager translation. We will develop static compiler based and dynamic rules-based or machine-learning based methods to eagerly translate only those blocks that have a high likelihood of being reached during future program execution. Second, explore algorithms and techniques to minimize the need for locks and increase available parallelism between the multiple compiler and application threads. Third, investigate techniques to understand other sources of startup overhead, including the fixed DBT initialization cost and indirect branch overhead before optimizations are activated, to further improve program startup performance.

VIII. Conclusions

Our goal with this work is to examine, understand and improve DBT system startup performance. We study the major sources of program overhead for short-running (startup) and long-running (steady-state) programs. We find that the cost of auxiliary tasks and user-level context switches are mostly incurred at program startup. Optimizations performed by the DBT system at startup enable execution to stay in the code cache context for longer periods during steady-state execution.

We found that block translations performed by a DBT system pose a large cost at program startup, and dominates the total execution time for some programs. We investigate the potential of novel techniques to reduce this translation overhead on multi-core machines. Our techniques use multiple parallel translation threads to eagerly or speculatively translate blocks before they are reached and stall program execution. We employ an offline approach to demonstrate the enormous potential of this technique to reduce translation costs, and find that prediction accuracy will need to be further improved to realize this potential with completely online techniques.

References

[1] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, 2005.

[2] Ho-Seop Kim and James E. Smith. Dynamic binary translation for accumulator-oriented architectures. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 25–35, 2003.

[3] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigel Zemach. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium-based systems. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 191–, 2003.

[4] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates. Fx!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, March 1998.

[5] Hewlett-Packard Development Company. HP ARIES binary compatibility and product support statement. accessed from http://www.hp.com/go/aries, September 2008.

[6] Apple Inc. Rosetta. accessed from http://www.apple.com/asia/rosetta, 2006.

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

[8] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The transmeta code morphing&trade; 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*, CGO '03, pages 15–24, 2003.

[9] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, 2007.

[10] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, 2005.

[11] Derek L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[12] Peter Feiner, Angela Demke Brown, and Ashvin Goel. Comprehensive kernel instrumentation via dynamic binary translation. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 135–146, 2012.

[13] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, 2000.

[14] Yukinori Sato, Tomoya Yuki, and Toshio Endo. Exanadbt: A dynamic compilation system for transparent polyhedral optimizations at runtime. In *Proceedings of the Computing Frontiers Conference*, CF'17, pages 191–200, 2017.

[15] Chaohao Xu, Jianhui Li, Tao Bao, Yun Wang, and Bo Huang. Metadata driven memory optimizations in dynamic binary translator. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, VEE '07, pages 148–157, 2007.

[16] Wei Hu, Jason Hiser, Dan Williams, Adrian Filipi, Jack W. Davidson, David Evans, John C. Knight, Anh Nguyen-Tuong, and Jonathan Rowanhill. Secure and practical defense against code-injection attacks using software dynamic translation. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 2–12, 2006.

[17] Mathias Payer and Thomas R. Gross. Fine-grained user-space security through virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '11, pages 157–168, 2011.

[18] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeÑo jvm. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 47–65, 2000.

[19] A. Ruiz-Alvarez and K. Hazelwood. Evaluating the impact of dynamic binary translation systems on hardware cache performance. In *2008 IEEE International Symposium on Workload Characterization*, pages 131–140, Sept 2008.

[20] Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker, and Stéphane Ducasse. Sista: Saving optimized code in snapshots for fast start-up. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes*, ManLang 2017, pages 1–11, 2017.

[21] Wenwen Wang, Pen-Chung Yew, Antonia Zhai, and Stephen McCamant. A general persistent code caching framework for dynamic binary translation (dbt). In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 591–603, 2016.

[22] Vijay Janapa Reddi, Dan Connors, Robert Cohn, and Michael D. Smith. Persistent code caching: Exploiting code reuse across executions and applications. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 74–88, 2007.

[23] Derek Bruening and Vladimir Kiriansky. Process-shared and persistent code caches. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 61–70, 2008.

[24] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. Hqemu: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 104–113, 2012.

[25] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™server compiler. In *Proceedings of the Symposium on Java Virtual Machine Research and Technology Symposium*, pages 1–12, 2001.

[26] DynamoRIO: Dynamic Instrumentation Tool Platform. accessed from http://www.dynamorio.org/home.html, April 2018.

[27] QEMU: The FAST! processor emulator. accessed from https://www.qemu.org/, April 2018.

[28] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.

[29] Arnaldo Carvalho de Melo. The new linux perf tools. Linux Kongress, 2010.