Indonesian Journal of Electrical Engineering and Computer Science
Vol. 10, No. 3, June 2018, pp. 1036~1044
ISSN: 2502-4752, DOI: 10.11591/ijeecs.v10.i3.pp1036-1044

Fine-grained Overhead Characterisation of Cross-ISA DBTO for Multicore Processor

Joo-On Ooi[1], Fawnizu Azmadi Hussin[2], Mohd. Nordin Zakaria[3]
[1]Department of Computer and Communication Technology, Universiti Tunku Abdul Rahman, Malaysia
[2]Department of Electrical and Electronic Engineering, Universiti Teknologi Petronas, Malaysia
[3]Department of Computer Information System, Universiti Teknologi Petronas, Malaysia

Article Info

Article history:

Received Jan 10, 2018
Revised Mar 16, 2018
Accepted Apr 2, 2018

Keywords:

Binary optimisation
Binary translation
Multicore
Multi-ISA processor
Multithreaded

ABSTRACT

The emergence of modern portable software, start to behaved hybrid short-long running combined applications, in which an active apps may invoked others to fulfill task requirements. Thus the implementation of Dynamic Translation and Optimisation (DBTO) into heterogeneous multicore system-on-chip (SoC) will require careful re-study, to ensure efficient usage of most available cores. In order to improve efficiency in supporting this Instruction Set Architecture (ISA) diversity of computing platforms, mix modes of statically and dynamically Binary Translation and Optimization system, or DBTO, need to utilize concurrent compilation techniques, to better service the combined applications processing. This research deep dived into finer-grained DBTO overhead analysis, to provide categorization and characterization of overhead sources in breakdown stages during concurrent instruction processing. A dual-engine of translation and optimization is constructed for finer managemnt of start-up overheads. Helper functions, i.e. LoadLink/StoreCondition (LL/SC) are derived from atomic instructions, to create multiple helper thread supported by multiple host cores, for better instruction translation and optimization operation concurrently. Our experiment platform, evaluated through PARSEC-3.0 benchmark suite, shows performance improvement approaching 2.0x for apps based programs and 1.25x for kernel based programs, for x86 to X86-64 emulation. This technique possess great potential and serve as research based platform for future binary translation technique development, including adaptive method.

Copyright © 2018 Institute of Advanced Engineering and Science.
All rights reserved.

Corresponding Author:

Joo-On Ooi,
Department of Computer and Communication Technology,
Universiti Tunku Abdul Rahman,
Jalan Universiti, Bandar Barat, 31900 Kampar, Malaysia.
Email: ooijo@utar.edu.my

1. INTRODUCTION

Dynamic Binary Translation (DBT) has been commonly used in cross-ISA process virtual machines [20] to enable system or application migration from one ISA to another [17]. Some popular application of this DBT includes Android emulator using QEMU [1] to develop ARM based code running on x86 machine [16, 17]. This fast emulation technique dynamically translates guest executables (eg. ARM binary) to native instructions on the host machine (eg. x86 server), and store the translated native code in cache memory to avoid re-translation [3, 8]. The translated code runs many times faster than the traditional interpretation approach, it can be further optimized through Dynamic Binary Optimization (DBO) process, in which redundant and superfluous instructions can be eliminated to reduce code size for faster code processing [2, 10]. However, to further speed up the emulation through generating highly optimized code has ever become more challenging since optimizations require longer translation time, which is a portion of the

system runtime, and potentially produced unreliable code if optimizations are not carefully tested [12]. The performance of a DBT-DBO emulator is greatly determined by turn-around time, in which execution time plays a main contributor. This overheads are non-negligible during dynamic process of transforming a piece of code into another, which caused the emulated program or system to pause progress momentarily [9]. Such overheads impact directly the overall system performance, which is not the only important metric, due to overheads from start-up or reactive code translation become significant as compare to relatively smaller overall overheads once an application has been executed. Past experiments done by researchers have shown that a typical DBT with DBO emulation process will goes through a series of common primitives despite the level at which it operates [5,18,20]. Some of the functions perform by the optimizer includes following, which may not necessarily in this sequence: (i) code profiling in order to detect hot regions, (ii) build regions, (iii) decode instructions, (iv) optimize regions, (v) code scheduling, (vi) code caching etc [2,10]. In this research , the essence of faster code translation in QEMU and rich optimizations possess by LLVM is combined into single hybrid translator-optimizer system, known as Dual-Engine DBTO, which is capable of handling multi-ISA guest code towards multi-ISA host processing, and produced both good translated code quality with relatively low translation overhead [20].

Furthermore, this paper look into overhead characterization in finer grained level, focusing on the specific parameters' delay time incurred during the activation of multiple helper threads by LL/SC instruction call, in which aimed to assist the simultaneous binary translation and optimization processing for this newly constructed DBTO system. Through reviewing the overhead process a set of formula for code transition overheads is derived, and this formula is validated through the simulation experiment using the DBTO system constructed.

Thus the main contributions of this research work are as follows:

a. A detail analysis on DBTO overheads, include classification, characterization and formulae derivations involving related influencing parameters.
b. A multi-threaded retargetable DBTO on multicores processor, capable for simultaneous binary translation and optimisation, is developed for hypothesis validation.
c. A novel fine-grained overhead characterization with formula derivation, caused by multiple helper threads creation through LoadLink and StoreCond instruction.
d. An experiment framework to validate the proposed fine-grained overhead characterization of the multiple helper threads supported DBTO system.

This research also intend to explore the possibility of other method(s) to reduce translation and optimization overheads incurred, as well as providing a useful platform for researcher, hopefully to produce a useful development tool or experimental prototype for embedded application, for instance Internet of Thing devices.

2. THEORETICAL BASIS

2.1 Dynamic Binary Translation ad Optimisation operation

During DBT, in order to achieve guest to host binary translation, DBT system uses a globally shared code cache, so that all executing threads shared a single code cache, and each guest block has only a single translated copy in the shared code cache [19]. All the threads will maintain one director that records the mapping from a guest code block to its corresponding translated host code region. An execution thread initially looks up the directory to locate the translated code region. Once not found, it activates the Tiny Code Generator (TCG) to translate the untranslated guest code block. As all the execution threads share the code cache and the mapping directory, QEMU uses a critical section to serialise all accesses to the shared structures, as shown in Figure 1 [15]. Typically TCG is meant to be lightweight optimizer, which provide an ideal platform for emulating short-running applications with few hot blocks, such as during the booting of an operating system. The problem observed from lower quality translated code during cross-ISA binary translation has encouraged the exploration and development of additional translation process, commonly known as Dynamic Binary Optimization, in which several "heavy" optimization passes being employed to improve the quality of translated code [10]. Hybrid type of DBT with DBO combined the advantages of faster guest code translation with potential for code optimization to yield reduced size binary code for faster host machine execution.

The translation and optimisation in DBTO operation aims to converting a code from one format to another, which cause various stages of overheads, mainly classified into native and instrumentation overheads [11]. Native overheads mainly due to startup and re-active overheads. Start-up overheads is the overheads occurred until the system reaches steady-state, where the vast majority of the executed code comes from the translated and optimized code cache. Re-active overheads are caused by re-translation and re-optimisation of regions of code that have been evicted from the translation cache, particularly in the case of a

multiprogrammed environment with shared translation cache [20]. All these overheads can cause significant slowdown to DBT processing, as shown in Figure 2, thus, reducing the total DBT and DBO overheads is of utmost importance [9].

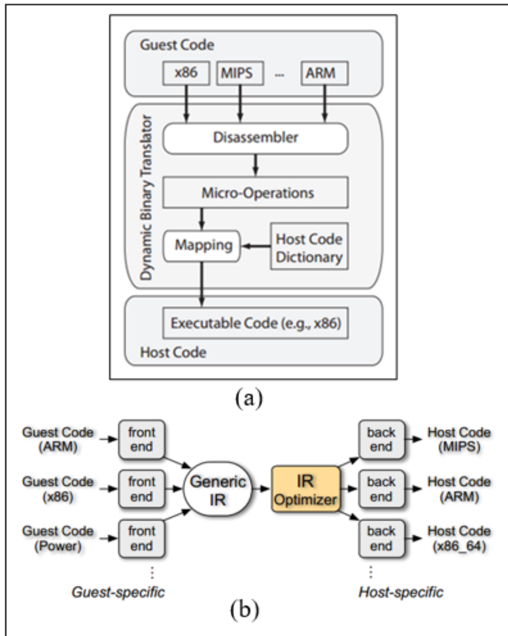

Figure 1. Dynamic binary translation operation, (a) overview block, (b) DBTO overview block disgram

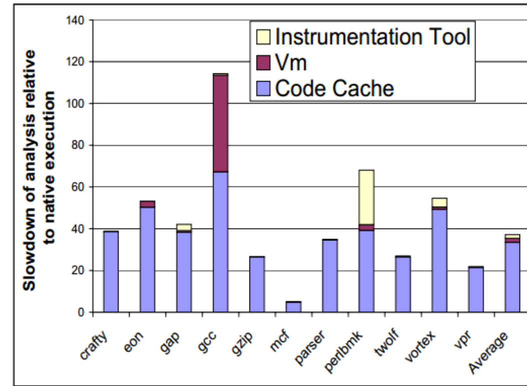

Figure 2. Slowdown analysis of DBT, in breakdown of fine-grained overheads [3]

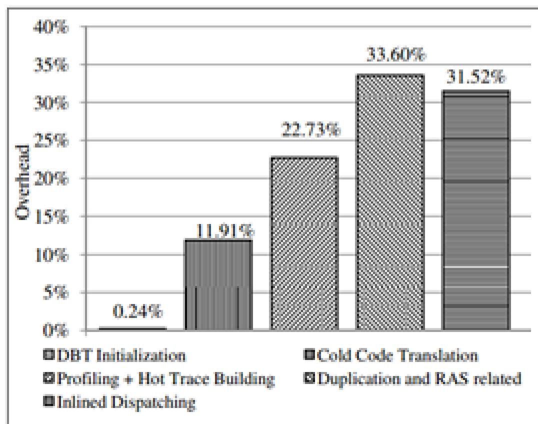

Figure 3. Breakdown of binary translation total overheads [32]

$$L_{\text{Initialization}} = \frac{DBT_{Native} - Native}{Native} \quad (1)$$

$$L_{\text{ColdTrans.}} = \frac{DBT_{DTraces} - (DBT_{PTrace} - DBT_{PTraces+CP})}{Native} \quad (2)$$

$$L_{\text{LT}} = DBT_{LT+Native} - DBT_{Native} \quad (3)$$

$$L_{\text{Prf}+HTBuild} = \frac{DBT_{DTraces} - (DBT_{PTrace} - S_{LT})}{Native} \quad (4)$$

$$L_{\text{Addr.Res}} = \frac{DBT_{PTraces+CP} - (DBT_{PTrace+CP+NAR})}{Native} \quad (5)$$

Figure 4. Equation for various fine-grained overheads [9]

2.2 Overhead Characterisation of DBTO

Overhead of the Dynamic Binary Translation and Optimization can be characterize by different fine-grain categories, namely initialization, cold code translation, profiling and hot trace building, and translated code execution; which will be further described. Initialisation overhead, the overhead incurred during loading of DBT system into memory [11]. The overhead is measured by executing the native code immediately after the DBT initilisation. The overhead formula as proposed by [9] is shown in Equation (1). Typical initialisation overhead is around 0.2%. The Cold Code Translation is the metric associated with the newly encountered code or not-yet-translated-code, reside in code cache. At the same time DBT also updates the previously translated blocks to branch into this newly translated code.

A DBT configuration which utilising persistant traces, known as DBTPTraces removes the profiling and hot trace building overhead, but still suffers from cold code translation overhead. To eliminate this cold code translation overhead, a newer configuration with code patching addition, known as DBTPTrace+CP (Code Patching) is constructed. This configuration load the persistant hot traces and the native code is patched with instructions to jump to the loaded traces, the patched code is then executed, thus eliminating cold code translation process.

Measurement for this overhead is shown in equation (2), with 11.91% as typical value. Profiling and Hot Trace Building:The application execution can be accelerated by optimizing frequently executed code, or known as hot code, by a mean of detection mechanism using runtime information, commonly using profiling technique to obtain hot traces, measurement as shown in Equation 4. This process involves two types of overheads, namely Profiling Instrumentation and Profiling Execution. Profiling Instrumentation overhead is the time required to perform the instrumentation, or tool infrastructure set up preparing for profiling code translation process. Whereas Profiling Execution overhead is the time spent during executing the profiling instructions [9]. The persistent trace loading overhead (Equation 3) is measured by comparing the execution time of specific benchmarking using DBTLT+Native and DBTNative configurations. This DBTPTraces approach removes the profiling and hot trace building overhead but adds the overhead to load the persistent traces from the target file, with typical values around 22.73%, as shown in Figure 3.Translated Code Execution: Ideally, eliminating the initialisation and translation overheads would make the translated code run at least as fast as the native code.

However, the translated code is not exactly the same as original code and the specialized hardware unavailability may cause extra overhead, which can be broken down into guest code emulation overhead, code cache control transfer overhead, Code Duplication and Return Address Stack (RAS) overhead, which will be described in the following section. Code Emulation overhead occurred due to the need to keep the original program behavior, thus translated code must emulate partial of the native instructions during execution, which involved more instruction emulations that potentially cause increment overheads. Code Cache Control Transfer will employs three modes of operations to perform the control transfer between traces and basic blocks inside the code cache, including code block chaining, code dispatching and inline dispatching. In typical situation, chaining does not incur extra overhead during the translated code execution. Due to inlined dispatching method, the Code Dispatcher is only called to resolve addresses from cold code.

However, cold code is not often executed and the overhead caused by calls to the Code Dispatcher is minimized. In this way, majority of the address resolution overhead occurs with the inlined dispatching, which is used within frequently executed hot code, and may represent a significant portion of address resolution overhead, in which its measurment is shown in Equation (5). Experiment done by other researcher [9] shows that the address resolution overhead cause by indirect jumps and return instructions accounts for approximately 31.5% of the total translation overhead, as shown in Figure 3. Researcher's experiment shows that some processor relies on RAS to efficiently predict the target address of return instructions, which form a norm by most modern microprocessors. However, return instructions cannot be executed inside the translated code and are normally emulated through indirect jump instructions,which greatly increases the indirect branch predictor cache pressure. Experiment indicated that typical DBTO overhead due to RAS is 33.6%, which accounted the highest weightage among all the related overheads.

3. RESEARCH METHOD

This research intend to provide concurrent binary translation usilising multithreading services on multicore, by constructing an infrastructure for atomic instruction which implemented in QEMU, chosen through hypothesis being made by the evidence of researcher's work that shown it's capability to perform parallel tasks processing through multiple compute units emulation [4, 6].

Through providing new TCG helpers act as sort of softmmu helpers, atomicity behavior can be guaranteed to some memory accesses. More specifically, the new softmmu helpers behave as LoadLink and StoreConditional instructions, and are called from TCG code by means of target specific helpers. The implementation heavily uses the software TLB together with a new bitmap that has been added to the ram_list structure which flags, on a per-CPU basis, all the memory pages that are in the middle of a LoadLink (LL), StoreConditional (SC) operation. LoadLink instruction is the instruction that reads the value from a shared memory location and stores the content into a register of the calling CPU. It also establishes a link and records the CPU with the accessed address (xaddr), to properly handle the subsequent SC operation. StoreConditional instruction is the instruction that writes to the address xaddr only if it belongs to an exclusive memory region (EMR) previously created by an LL. The SC is not always successful since another CPU can nullify the EMR by writing or reading to it. Since all these pages can be accessed directly through the fast-path and alter a vCPU's linked value, the new bitmap has been coupled with a new TLB flag for the

TLB virtual address which forces the slow-path execution for all the accesses to a page containing a linked address. This new slow-path implemention demonstrates the following features:

a. The LL behaves as a normal load slow-path, except for clearing the dirty flag in the bitmap. The cputlb.c code while generating a TLB entry, checks if there is at least one vCPU that has the bit cleared in the exclusive bitmap, in that case the TLB entry will have the EXCL flag set, thus forcing the slow-path. The TLB cache of all the other vCPUs is flushed to ensure that all the vCPUs will follow the slow-path for that page. The LL will also set the linked address and size of the access in a vCPU's private variable. After the corresponding SC, this address will be set to a reset value.
b. The SC can fail by returning 1, or succeed by returning 0. It has to come always after a LL and has to access the same address 'linked' by the previous LL, otherwise it will fail. If in the time window delimited by a legit pair of LL/SC operations another write access happens to the linked address, the SC will fail.

In theory, the provided implementation of TCG LoadLink/StoreConditional can be used to properly handle atomic instructions on any processor architecture. During implementation work, two new instructions are created into existing 132 TCG ops instruction set, mainly to handle load linking and conditional store operation between related registers and host memory. This two instructions are introduced as helper instructions known as helper_ldlink_name and helper_stcond_name.

Operations of One of the major problems when dealing with multi-threaded programs is the occurrence of race conditions. In the context of this work, a race condition can be associated to an inconsistency of the whole machine state, which is in charge of translating atomic instructions. The direct negative result of such a state is the failure of a StoreConditional (SC) operation that should have succeeded, or even worse, the success of a SC operation that had to fail. In the subsequent sections, all critical points that result in race conditions are explored, where the implemented approach is also documented. Updates of the exclusive bitmap can lead to inconsistencies due to the out-of-order execution of load/store operations as seen, for instance, on ARM architectures [16]. For this reason all accessors to such a bitmap are atomic, an outcome that is possible by means of host atomic instructions. It is important to note, that this can be possibly achieved only in the case where bitmap accessors are QEMU functions and not implemented through TCG generated code. In fact, other guest CPUs, different from the one issuing the LL, could have already generated TLB entries for the same page, forcing the execution to follow the fast-path (as shown in in Figure 5). To avoid this dangerous behaviour, TLB entries of these CPUs will be flushed, forcing them to recreate the TLB entry that covers the page in the EMR. This flush request will also prevent race conditions that are related to the delayed new state propagation of the exclusive bit. Our implementation also ensure the evaluations and updates of the EMRs have been safeguarded using a mutex, due to updating this structure is not possible with a single atomic instruction. Another related aspect that requires additional caution, relates to the actual memory accesses made by the LL and SC instructions. More specifically, the results on memory brought by these instructions has also to be done jointly with the update of the EMR values. The Listings 1, 2 and 3 represent respectively the algorithms for LL, SC and normal store access. In these examples, the critical region is delimited by two calls LOCK and UNLOCK. LoadLink as in Listing 1, only works as long as the normal local is done inside the critical section, otherwise the loaded value can be potentially updated by another CPU, which might or might not be inside the critical region. For the same reason, the SC operation (Listing 2) has also to rely on the same critical region to be consistent with the rest of the atomic instruction emulation. Without entering the critical region, it can potentially declare the operation as successful (returning 0), but performing the store after another CPU modified the value. Similarly, the store operation (Listing 3) enters the critical region to check for a possible conflict in EMR, but also to perform the regular access.

```
input: int y, int z, int x
output: int
begin:
   LOCK()
   CPU[x].EMR ← [y, y + z]
   ret ← load(y,z)
   UNLOCK()
   return ret
end
```

```
input: int y, int z, int x,
      int val
output: int
begin:
   LOCK()
   if CPU[x].EMR = [y, y + z]
      store(y,z,val)
      ret ← 0
   else
      ret ← 1
   end
   UNLOCK()
   return ret
end
```

```
input: int y, int z, int val
begin:
   LOCK()
   for each CPU
      if CPU.EMR overlaps [y, y + z]
         CPU.EMR ← NULL
      end
   store(y, z, val)
   UNLOCK()
end
```

Listing 1 (left): LoadLink pseudo code, load ( ) denotes a plain load from memory of size z
Listing 2 (middle): StoreCond pseudo code, store ( ) denotes a plain store to memory of size z
Listing 3 (right): Plain write access trapped by the slow-path

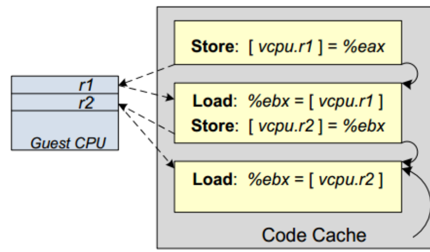

Figure 5. Code region transition between guest CPU registers and code cache

The overheads involved during this LL/SC operation includes helper function calls and code transition. During the QEMU system-mode emulation process, the LoadLink instruction operation takes place while the code transition is in process. Upon activation, helper function which comes in a piece of C code will be called from translated code, so to store the current address and load the value. Firstly a bit in the exclusive bitmap will be set to enforce slow path, which means helper function call for achieving multiple helper thread. DBT need to set the link address of guest CPU memory by first obtaining a lock for the target guest CPU memory segment to access the critical region, then the address size is determined, followed by setting the address range into the targeted vCPU thread register. Upon conditional store instruction activation, all vCPUS are halted, DBT to check if this process has been interrupted since last LL call, through checking the TLB table by comparing current address and value with the saved copies in the TLB table, if unchanged store process is allowed and success operation reported, the cpu states from current guest CPU will then be save into code cache. Else the TLB entry table needs to be updated through TLB flushing for all vCPUs. This process is repeated for different basic blocks inside the code cache, thus to generate multiple helper function thread to assist in binary translation as well as optimization process. Based on the LoadLink and StoreCond operation described previously, the overhead of code transition due to Load Linking process can be modeled by constructing overhead formula. As we have described previously, Load Linking process is affected by process including critical region inside memory locking, address and it's size setting, flushing TLB, loading cpu state and unlocking critical region of memory. Thus the Load link overhead can be derived as below:

$$L_{\text{load_link}} = T_{mem_lock} + T_{link_addr_set} + T_{addr_size_set} + T_{TLB_flush} + T_{cpu_state_load} + T_{mem_unlock}$$

The overhead of code transition due to Store Conditional process, influenced by halting all vCPUs, comparing values, and saving values, is derived:

$$L_{\text{store_cond}} = T_{vCPU_halt} + T_{cmp} + T_{save_value}$$

Thus the code transition overhead due to LoadLink and store conditional process is then given:

$$L_{\text{code_trans.}} = T_{load_link} + T_{store_cond}$$

4. RESULTS AND ANALYSIS

Experiment is done through simulation of selected PARSEC-3.0 benchmark programs [14], compiled with gcc version 4.8.3,as depicted in Table 1. All performance evaluation is done on a system with one 1.7 GHz quad-core Intel Core-i7 processor and 4 GBytes main memory. The operating system is 64-bit Ubuntu 14.04 LTS Linux with kernel version 3.19.0-33-generic. The selected PARSEC 3.0 [15] benchmark programs are evaluated with the simlarge input sets resemble real inputs using larger problem size of data sets, for x86-32 guest ISA on the x86-64 host platform. All the selected benchmark programs are parallelized with the Pthread model and compiled for respective guest ISAs with PARSEC default compiler optimization and SIMD enabled. The benchmarks simulation performance is compared through using simlarge inputs between three different configurations: (i) Hybrid-QEMU with single-thread mode, denote as Hybrid-Q-s, and (ii) Hybrid-QEMU with multi-thread mode, denote as Hybrid-Q-m. During experiments, atomic instructions are emulated with lightweight memory transactions, for all the experiment configurations, so that the benchmarks can be emulated correctly.

Table 1. Selected Benchmark programs features

Program	Cat.	Application Domain	Parallelization		Working Set	Data Usage	
			Model	Granularity		Sharing	Exchange
blackscholes	apps	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	apps	Computer Vision	data-parallel	medium	medium	high	medium
canneal	kernel	Engineering	unstructured	fine	unbounded	high	high
ferret	apps	Similarity Search	pipeline	medium	unbounded	high	high
streamcluster	kernel	Data Mining	data-parallel	medium	medium	low	medium
vips	apps	Media Processing	data-parallel	coarse	medium	low	medium

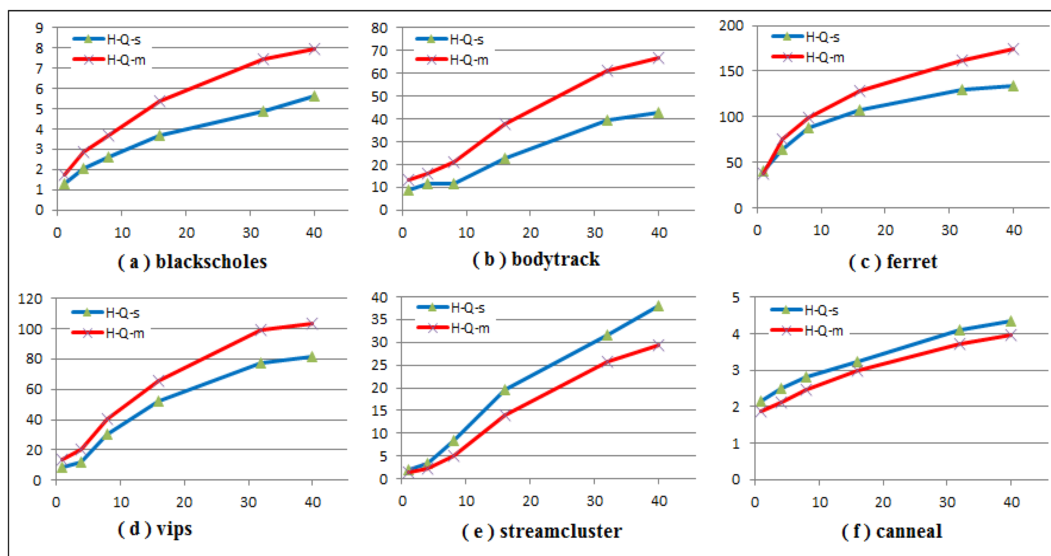

Figure 6. PARSEC-3.0 benchmark results of x86 to x86-64

4.1. Static and Dynamic Binary Translation performance

It is observed that translation time by single thread support were seen to be much shorter than those multithreaded support for all apps based benchmark programs, as seen in Figure 6(a) to (d). Whereas for kernel based programs, i.e. streamcluster and canneal, translation time for multithreaded support were seen shorter or at most closely similar to single thread support, as seen in Figure 6(e) and (f). The relatively poorer translation performance of multithreaded support for benchmark apps were mostly due to accumulated overheads incurred by thread contention due to multiple thread supporting initializations stage, which can result in significant performance degradation, despite the fact that the concurrent execution supported by multithreading should result in translation time reduction. In the other way round, improved translation with optimization time performance was observed for kernel based programs for H-Q-m, due to they are either fine-granular or coarse-granular programs geared for parallelism in nature, thus benefited through multithreaded process support scheme during the binary translation process.The apps based programs are typically short running program, thus they will gain beneficial from DBT, which can be observed from the shorter elapsed time taken for H-Q-s. Whereas kernel based apps are towards longer running program, in which they will be benefited from DBT and DBO.

4.2. Concurrent Dynamic Binary Translation performance

As shown in Figure 6 for almost all PARSEC-3.0 benchmark programs, the increment of the elapsed time is seen to be reduced with the growing of the number of worker threads. It is observed that this translation time increment gradually reach saturation stage when number of worker thread exceed 32 threads. This phenomenon is due to the contribution from our built-in concurrent features, which amortise the start-up overheads at certan period after start-up process, eventually reaching steady state when number of of activated worker threads reaching 40.

4.3. Program Start-up overhead analysis

During the start-up stage of guest to host ISA binary translation, the average elapsed time for the helper threads spent in critical sections will increase significantly due to the helper threads contending for critical section within the QEMU dispatcher where the serialization lengthens the wait time. The delay is further worsten by critical section access wait time and branch target mapping directory lookup time. Furthermore this latency which increased from such serialisation is greater than the reduced execution time gained from incremental helper threads which assist in binary translation process. Thus this latency has generated high overheads which dominates the total translation time and hence the overall execution time, eventually causes the poor performance of the parallel PARSEC-3.0 benchmarking for the selected apps.

5. CONCLUSION

This paper presented detail fine-grained analysis of concurrent dynamic translation and optimization incurred on our newly constructed Dual-Engine DBTO architecture, with multi-threaded retargetable capability running on multicores processor. Experiments has shown that such multi-threaded hybrid translation and optimization approach can achieve relatively lower translation overhead and yet with good translated code quality on the target binary applications, especially for kernels based programs. In this experiment, the H-Q-m supported by multiple threads for binary translation processing, is more efficient for kernel based applications, as shown by up to 1.25x speedup of multithread versus single thread. Whereas apps based program are more beneficial through single threaded supported binary translation with up to 1.8x speedup versus multiple threads, as also reported in our previous paper [20]. We foresee the great potential of utilising the multithread technique for assisting binary translation and optimisation process, both for short running and long running program analysis.

ACKNOWLEDGEMENTS

The author would like to thank Assoc. Prof Dr. Fawnizu, Dr. Nordin and their students for great support and valuable comments.

REFERENCES

[1] F. Bellard, *"QEMU, a fast and portable dynamic translator,"* in USENIX Annual Technical Conference, pp. 41-46, 2005.
[2] C. Lattner and V. Adve, *"LLVM: A compilation framework for lifelong program analysis & transformation,"* in Proceeding CGO, 2004.
[3] Uh GR, Cohn R, Yadavalli, B., Peri, R. and Ayyagari, R., *"Analyzing dynamic binary instrumentation overhead,"* in WBIA Workshop at ASPLOS, Oct 2006.
[4] Ding J. H., Chang P.C., Hsu W.C., and Chung Y.C., "PQEMU: Aparallel system emulator based on QEMU," in Parallel and Distributed Systems (ICPADS), *2011 IEEE 17th In'l Conference,* 7 Dec 2011, pp. 276-283.
[5] Jeffery A., "Using the LLVM compiler infrastructure for optimised, asynchronous dynamic translation in QEMU," Master's thesis, University of Adelaide, Australia, 2009.
[6] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zhang. "COREMU: a scallable and portable parallel full-system emulator," in *Proc. PPoPP,* 2011.
[7] Baraz, Leonid, et al. *"IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems,"* Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2003.
[8] Dehnert, James C., et al. *"The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges,"* Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, IEEE Computer Society, 2003.
[9] Borin, Edson, and Youfeng Wu. *"Characterization of DBT overhead."* Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, 2009.
[10] J. Lu, H.Chen, P.-C.Yew, and W.-C.Hsu, "Design and implementation of a lightweight dynamic optimization system," *Journal of Instruction-Level Parallelism*, 6:1–24, 2004.
[11] C.K.Luk, R.Cohn, R.Muth, H.Patil, A.Klauser, G.Lowney, S.Wallace, V.Reddi, and K. Hazelwood, "*Pin: Building customized program analysis tools with dynamic instrumentation*," In Proc. PLDI, 2005.
[12] K. Scott, N.Kumar, B.R.Childers, J.W.Davidson, and M.L.Soffa, *"Overhead reduction techniques for software dynamic translation,"* In Proc. IPDPS, pages 200–207, 2004.
[13] S. Sridhar, J.S.Shapiro, E.Northup, and P.P.Bungale, *"HDTrans: an opensource, low-level dynamic instrumentation system,"* In Proc.VEE, pages 175–185, 2006.

[14] Bienia, Christian, et al. *"The PARSEC benchmark suite: Characterization and architectural implications,"* Proceedings of the 17th international conference on Parallel architectures and compilation techniques. ACM, 2008.

[15] Cifuentes, Cristina, and VishvMalhotra. *"Binary translation: Static, dynamic, retargetable?" Software Maintenance 1996*, Proceedings.,International Conference on. IEEE, 1996.

[16] Chen, Jiunn-Yeu, et al., *"A static binary translator for efficient migration of ARM-based applications,"* Workshop on Optimizations for DSP and Embedded Systems. 2008.

[17] Kondoh G, Komatsu H., *"Dynamic binary translation specialized for embedded systems,"* ACM Sigplan Notices. 2010 Jul 1;45(7):157-66.

[18] Shen, Bor-Yeh, et al., *"An LLVM-based hybrid binary translation system,"* Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on. IEEE, 2012.

[19] Liu, Chia-Lun, et al. "Dynamically Translating Binary Code for Multi-Threaded Programs Using Shared Code Cache," *Journal of Electronic Science and Technology* no. 4 (2014): 434-438.

[20] Ooi, J. O., Hussin, F. A. B., &Zakaria, N., (2016, December). *"Dual-Engine Cross-ISA DBTO Technique Utilising MultiThreaded Support for Multicore Processor System,"* In Embedded Multicore/Many-core Systems-on-Chip (MCSoC), 2016 IEEE 10th International Symposium, December 2016, pp. 257-264.

BIOGRAPHIES OF AUTHORS

Joo-On Ooi received his MSc and Bachelor degree in Electrcal and Electronics Engineering from Nanyang Technological University, Singapore, in 2004 and 1999 respectively. From 2002 until 2011 he has been involving in Application Engineering in semiconductor field, covering microprocessor, microcontroller and memory devices. Prior to this he has been involving in satellite sub-system design in Satellite Engineering center, NTU, Singapore. Since April 2011 he has been the faculty member in the Department of Computer and Communication Technology, Universiti Tunku Abdul Rahman, Malaysia. His research interests are in the areas of multicore processor, runtime system, real-timeprocessing, compiler binary translation and optimisaion. Currently he is a member of IEEE Circuit and System Society.

Fawnizu Azmadi Hussin received his PhD degree in electrical and electronic under Monbukagakusho scholarship of MEXT from Nara Institute of cience and Technology, Japan.While currently served as Associate Professor at the Electrical and Electronics Engineering Department, and member of the Centre for Intelligent Signal and Imaging Research,Universiti Teknologi Petronas (UTP); he was the Deputy head of EE Dept. from 2012-2013, and also Progam Manager of MSc programme. He assumed Director of UTP Strategic Alliance Office since Oct. 2014, and was past-chair of IEEE Circuit and System society Malaysia chapter during 2013-2014. His research interests are in the areas of VLSI design and testing, particularly SoC Design-for-Test and scheduling, NoC interconnect, low-power VLSI and FPGA's algorithm and architecture implementation/optimization.

Mohamed Nordin Zakaria received his PhD degree in Computer Science from Universiti Sains Malaysia, Malysia. Currently Dr. Nordin assumed as Head of High Performance Computing Centre (HPCC), UTP, Malaysia. His research interests are in the areas of computer graphics, evolutionary algorithm and scheduling for large scale computing infrastructure.