ISSN 0361-7688, Programming and Computer Software, 2012, Vol. 38, No. 3, pp. 119–126. © Pleiades Publishing, Ltd., 2012.

# Background Optimization in Full System Binary Translation[1]

**R. A. Sokolov [a] and A. V. Ermolovich [b]**

[a] *MCST CJSC, ul. Krasnosel'skaya Nizhnyaya 35, 105066, Moscow, Russia*

[b] *Intel CJSC, ul. Krylatskaya 17, 121614 Moscow, Russia*

*e-mail: roman.a.sokolov@gmail.com, karbo@pvk13.org*

Received November 30, 2011

**Abstract**—Binary translation and dynamic optimization are widely used to provide compatibility between legacy and promising upcoming architectures on the level of executable binary codes. Dynamic optimization is one of the key contributors to dynamic binary translation system performance. At the same time it can be a major source of overhead, both in terms of CPU cycles and whole system latency, as long as optimization time is included in the execution time of the application under translation. One of the solutions that allow to eliminate dynamic optimization overhead is to perform optimization simultaneously with the execution, in a separate thread. In the paper we present implementation of this technique in full system dynamic binary translator. For this purpose, an infrastructure for multithreaded execution was implemented in binary translation system. This allowed running dynamic optimization in a separate thread independently of and concurrently with the main thread of execution of binary codes under translation. Depending on the computational resources available, this is achieved whether by interleaving the two threads on a single processor core or by moving optimization thread to an underutilized processor core. In the first case the latency introduced to the system by a computational intensive dynamic optimization is reduced. In the second case overlapping of execution and optimization threads also results in elimination of optimization time from the total execution time of original binary codes.

**DOI:** 10.1134/S0361768812030073

## 1. INTRODUCTION

Technologies of binary translation and dynamic optimization are widely used in modern software and hardware computing systems [1]. In particular, dynamic binary translation systems (DBTS) comprising these technologies serve as a solution to provide compatibility between widely used legacy and promising upcoming microprocessor architectures on the level of executable binary codes. In the context of binary translation these architectures are usually referred to as source and target, correspondingly.

DBTSs execute binary codes of source architecture on top of instruction set (ISA) incompatible target architecture hardware. They perform translation of executable codes incrementally (as opposed to whole application static compilation) interleaving it with execution of generated translated codes. One of the key requirements that every DBTS has to meet is that the performance of execution of source codes through binary translation is to be comparable or even outperform the performance of native execution (when executing them on top of source architecture hardware).

Optimizing translator is usually employed to achieve higher DBTS performance. It allows to generate highly efficient target architecture codes fully utilizing all architectural features introduced to support binary translation. Besides, dynamic optimization can benefit from utilization of actual information about executables behavior which static compilers usually don't possess.

At the same time dynamic optimization can imply significant overhead as long as optimization time is included in the execution time of application under translation. Total optimization time can be significant but will not necessarily be compensated by the translated codes speed-up if application run time is too short.

Also, the operation of optimizing translator can worsen the latency (i.e., increase pause time) of interactive application or operating system under translation. By latency is meant the time of response of computer system to external events such as asynchronous hardware interrupts from attached I/O devices and interfaces. This characteristic of a computer system is as important for the end user, operation of hardware attached or other computers across network as its overall performance. Full system dynamic binary translators have to provide low latency of operation as well. Binary translation systems of this class target to implement all the semantics and behavior model of source architecture and execute the entire hierarchy of system-level and application-level software including

[1] The article is published in the original.

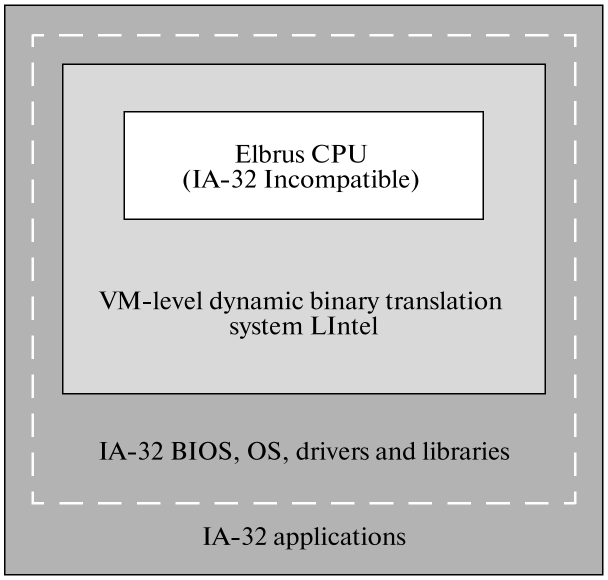
Elbrus CPU
(IA-32 Incompatible)
VM-level dynamic binary translation system LIntel
IA-32 BIOS, OS, drivers and libraries
IA-32 applications

**Fig. 1.** VM-level dynamic binary translation system LIntel.

BIOS and operating systems. They exclusively control all the computer system hardware and operation. Throughout this paper we will also refer this type of binary translation systems as virtual machine level (or VM-level) binary translators (as opposed to application-level binary translators).

One recognized technique to reduce dynamic optimization overhead is to perform optimization simultaneously (concurrently) with the execution of original binary codes by utilizing unemployed computational resources or free cycles. It was utilized in a number of dynamic binary translation and optimization systems [2, 8]. We will refer this method as background optimization (as opposed to consequent optimization, when optimizing translation interrupts execution and utilizes processor time exclusively unless it completes).

The paper describes implementation of background optimization in a VM-level dynamic binary translation system. This is achieved by separating of optimizing translation from execution flow into an independent thread which can then concurrently share available processing resources with execution thread. Backgrounding is implemented whether by interleaving the two threads in case of a single-core (single processor) system or by moving optimization thread to an unemployed processor core in case of a dual-core (dual processor) system. In the first case the latency introduced to the system by the "heavy" phase of optimizing translation is reduced. In the second case, overlapping of execution and optimization threads also eliminates the time spent in dynamic optimization phase from the total run time of the original application under translation.

The specific contributions of this work are as follows:

• implementation of multithreaded infrastructure in a VM-level dynamic binary translation system;

• single processor system targeted implementation of background optimization technique where processor time sharing is implemented by interleaving optimizing translation with execution of original binary codes;

• dual processor system targeted implementation of background optimization technique where optimizing translation is being completely offloaded onto underutilized processor core.

The solutions described in the paper were implemented in the VM-level dynamic binary translation system LIntel, which provides full system-level binary compatibility with Intel IA-32 architecture on top of Elbrus architecture [9, 10] hardware.

## 2. LINTEL

Elbrus is a VLIW (Very Long Instruction Word) microprocessor architecture. It has several special features including hardware support for full compatibility with IA-32 architecture on the basis of transparent dynamic binary translation.

LIntel is a dynamic binary translation system developed for high performance emulation of Intel IA-32 architecture system through dynamic translation of source IA-32 instructions into wide instructions of target Elbrus architecture (the two architectures are ISA-incompatible). It provides full system-level compatibility meaning that it is capable of translating the entire hierarchy of source architecture software (including BIOS, operating systems and applications) transparently for the end user (Fig. 1). As is noted above, LIntel is a co-designed system (developed along with the architecture, with hardware assistance in mind) and heavily utilizes all the features of architecture introduced to support efficient IA-32 compatibility.

In its general structure LIntel is similar to many other binary translation and optimization systems described before [11, 13] and is very close to Transmeta's Code Morphing Software [14, 15]. As any other VM-level binary translation system, it has to solve the problem of efficient sharing of computational resources between translation and execution of original binary codes.

### *2.1. Adaptive Binary Translation*

LIntel follows adaptive, profile-directed model of translation and execution of binary codes (Fig. 2). It includes four levels of translation and optimization varying by the efficiency of the resulting Elbrus code and the overhead implied, namely: interpreter, non-optimizing translator of traces and two optimizing translators of regions. LIntel performs dynamic profiling to identify hot regions of source code and to apply reasonable level of optimization depending on execut-

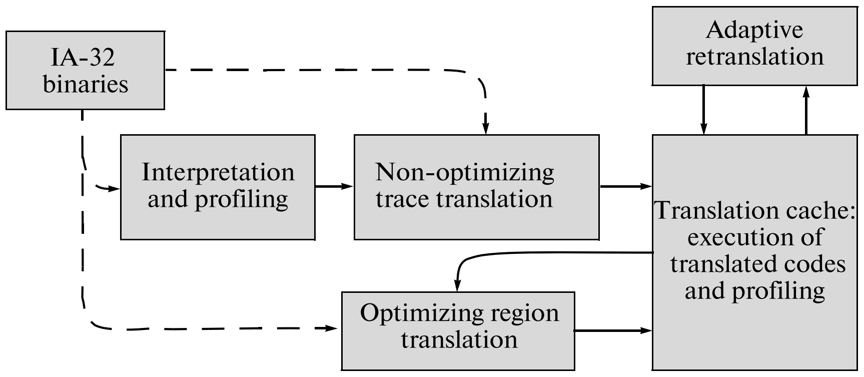
IA-32 binaries
Adaptive retranslation
Interpretation and profiling
Non-optimizing trace translation
Translation cache: execution of translated codes and profiling
Optimizing region translation

**Fig. 2.** Adaptive binary translation.

able codes behavior. Translation cache is employed to store and reuse generated translations throughout execution. Run-time support system controls the overall binary translation and execution process.

When the system starts, interpreter is used to carefully decode and execute IA-32 instructions sequentially, with attention to memory access ordering and precise exception handling. Non-optimizing translation is launched if execution counter of a particular basic block exceeds specified threshold.

Non-optimizing translator builds a *trace* which is a semantically accurate mapping of one or several contiguous basic blocks (following one path of control) into the target code. The building blocks for the trace are templates of the corresponding IA-32 instructions, where template is a manually scheduled sequence of Elbrus wide instructions. After code generation and additional corrections like actual constants and address values patching the trace is then stored into the translation cache. Trace translator produces native code without complex optimizations and focuses more on fast translation generation rather than code efficiency. It improves start-up time significantly as compared to interpretation. At the same time non-optimizing translation is only reasonable for executable codes with low repetition rate.

Traces are instrumented to profile hot code for O0-level optimizing translation. The unit of optimizing translation is a *region*. In contrast to traces, regions can combine basic blocks from multiple paths of control providing better opportunities for optimization and speculative execution (which is an important source of instruction level parallelism for VLIW processors).

O0-level translator is a fast region-based optimizer that performs basic optimizations implying low computation cost, including peephole, dead-code elimination, constant propagation, code motion, redundant load elimination, superblock if-conversion and scheduling.

Strong O1-level region-based optimizer is on the highest level of the system. The power of this level is comparable with high-level language optimizing compilers.[2] It applies advanced optimizations such as software pipelining, global scheduling, hyperblock if-conversion and many others, as well as utilizes all the architectural features introduced to support binary optimization and execution of optimized translations.

Region translations are stored in the translation cache as well. Profiling of regions for O1-level optimization is carried out by O0-level translations.

Optimized translations not always result in performance improvement. Unproven optimization time assumptions can cause execution penalty. These include incorrect speculative optimizations, memory mapped I/O access in optimized code (where I/O access is not guaranteed to be consistent due to memory operations merge and reordering), etc. Correctness of optimizations is controlled by the hardware at runtime. Upon detecting a failure, retranslation of the region is launched applying more conservative assumptions depending on failure type.

Figure 3 compares average translation cost of one IA-32 instruction and the performance of translated codes for different levels of optimization. Adaptivity aims at choosing appropriate level of optimization throughout the translation and execution process to maintain overhead/performance balance.

Figure 4 shows translation and execution time distribution for SPEC2000 tests running under Linux (operating system is being translated as well). While translated codes are executed most of the tests' runtime, optimizing translation overhead is significant and equals to 7% on average.

### *2.2. Asynchronous Interrupts Handling*

One of the run-time support system functions is to handle incoming external (aka asynchronous) interrupts. The method of delayed interrupt handling allows to improve the performance of binary translated

[2] In fact, O0/O1 notation of LIntel's binary optimizers corresponds to conventional 02/O3-O4 optimization levels of language compilers.

| | Cycles per one source instruction translation | Translated code performance |
|---|---|---|
| Non-optimizing translation | 1600 | 0.18 |
| O0 optimization | 30000 | 0.58 |
| O1 optimization | 1000000 | 1.0 |

**Fig. 3.** Average translation overhead per one IA-32 instruction and the performance of translated codes (normalized to O1).

code execution and interrupt handling by specifying exactly where and when a pending interrupt can be handled. When interrupt occurs, interrupt handler only remembers this fact by setting corresponding bit in the processor state register and returns to execution. Interpreter checks for pending interrupts before next instruction execution. Due to efficiency reasons, non-optimized traces only include such checks in the beginning of basic blocks. Optimizing translators inject special instructions in particular places of a region code (where execution context is guaranteed to be consistent) that check for pending interrupts and force execution flow to leave region and switch to interrupt handler if needed.

This method of pending interrupt checks arrangement simplifies planning and scheduling of translated codes as there is no need to care about correct execution termination and context recovery at arbitrary moments of time. At the same time it allows LIntel to respond reactively enough to external events.

The bottleneck in this scenario is the presence of optimizing translation phase. If an interrupt occurs when optimization is in progress, it has to wait for optimization phase completion to be handled (Fig. 5). Due to computational complexity of optimizations employed, optimizing translation can consume significant amount of processor time and as such, the delay of response of the system to an external event can be noticeable (see evaluation in Section 3.2).

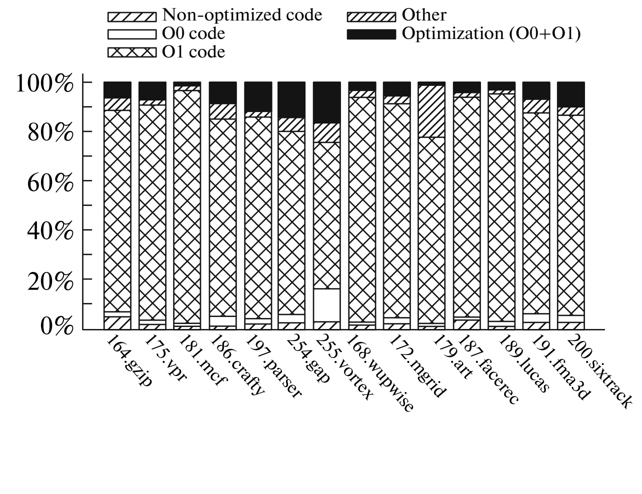

**Fig. 4.** Profile of binary translation in case of consecutive dynamic optimization.

## 3. BACKGROUND OPTIMIZATION

To overcome the problems of performance overhead and latency caused by optimizing translation, the method of back-ground optimization was employed in LIntel.

The concept of background optimization implies performing optimizing translation phase concurrently (or pseudo-concurrently) with the main binary translation flow of execution of original binary codes. Application-level binary translators usually implement this by utilizing native operating system's multithreading interface and scheduling service to perform optimization in a separate thread. VM-level binary translation systems require internal implementation of multithreading to support background optimization.

In this section we describe implementation of background optimization in the VM-level DBTS LIntel. Two cases are considered: in the first case LIntel operates on top of a single-core target platform system; in the second case there are two cores available for utilization.

SPEC2000 tests are used to demonstrate the effect of background optimization implementation.

### *3.1. Execution and Optimization Threads*

A multithreaded execution infrastructure was implemented in LIntel, with optimizing translation capable of running independently in a separate thread, which enabled execution and optimization threads concurrency. Execution thread activity includes the entire process of translation and execution of original binary codes, except for optimizing translation (of both O0 and O1 levels), i.e.: interpretation, non-optimizing translation, run-time support and execution itself. Optimizing translator is run in a separate optimization thread when new region of hot code is identified by the execution thread. When optimization phase completes, generated translation of the region is returned to the execution thread, which places it into the translation cache.

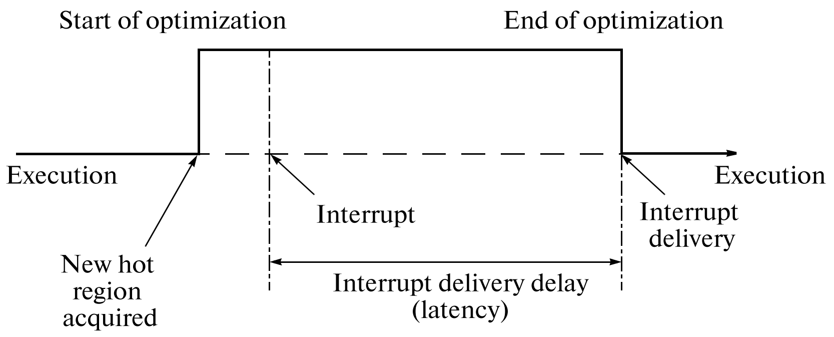

**Fig. 5.** Asynchronous interrupt delivery delay (latency) due to optimizing translation.

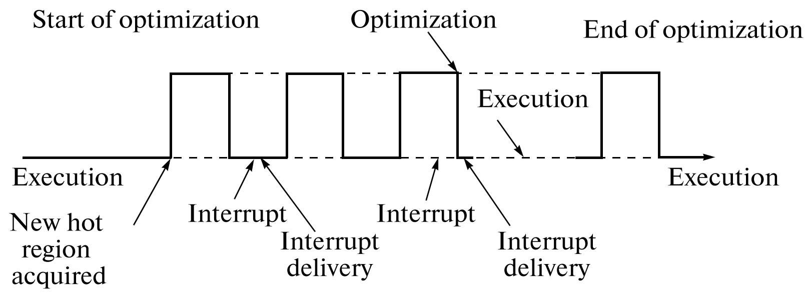

**Fig. 6.** Asynchronous interrupt delivery in case of interleaved background optimization.

During the region optimization phase corresponding original codes are being executed either by interpretation or by previously translated codes of lower levels of optimization. Selection of new hot regions for optimization will not be launched unless current optimization activity completes.

By the end of optimization, memory pages that contain a source code of the region under optimization can get invalidated (due to DMA, self-modification, etc.). As such, before placing optimized translation of the region into the translation cache, execution thread must check region's source code consistency and reject the region if verification fails. This routine is assisted by the memory protection monitoring subsystem (introduced in the Elbrus hardware to support binary translation [16]) which controls source and translated (as well as translations-in-progress) codes coherency.

Separation of execution and optimization threads allows to schedule them across available processing resources in the same way as multitasking operating systems schedule processes and threads. By now, two simple strategies of processor time sharing were implemented in LIntel enabling optimization backgrounding for single-core and dual-core systems.

### *3.2. Background Optimization in a Single-Core System*

In case of a single-core system background optimization is implemented by interleaving of execution and optimization threads. Throughout optimizing translation of a hot region processor switches between the two threads. Scheduling is triggered by interrupts from internal binary translation dedicated timer "invisible" for executable codes under translation. Each thread is assigned a fixed time slice. When execution thread is active, incoming external interrupt has a chance to be handled without having to wait for region optimization to complete (Fig. 6). If there are no hot regions pending for optimization, execution thread fully utilizes the processor core.

To demonstrate single-core background optimization approach, a simple strategy of processor time sharing was chosen when both threads have equal priority, with equal time slices assigned (meaning that optimization thread's processor utilization is 50%, in contrast to 100% utilization when optimizing consequently). As seen from Fig. 7, interleaving of execution and optimization improves interrupt delivery time significantly.

At the same time, as Fig. 8 demonstrates, this approach tend to degrade binary translation performance.

Degradation can be explained by the fact that hot region optimization phase now lasts longer. As a result,

| | Consecutive optimization | Interleaved (background) optimization |
|---|---|---|
| O1 phase mean time | 1.54 s | 3 s |
| O1 phase max time, $T_{01\ max}$ | 8.8 s | 29.5 s |
| interrupt delivery mean time with no optimization in progress | 54 μs | |
| interrupt delivery max time (with O1 phase in progress) | 8.8 s ($T_{01\_max}$) | 1.7 ms |

**Fig. 7.** Interrupt delivery time (CPU frequency = 300 MHz; thread time slice = 50000 cycles). O1-level optimization time is used as a reference as this phase consumes a greater number of processor cycles per source instruction as compared to O0-level optimization.

optimized translations injection into execution is being delayed, meanwhile source binary codes are being executed non-optimized (or interpreted). Additional overhead comes with context switching routines.

Basically, single-core background optimization implementation is not of high priority currently. At the same time we believe that it is possible to improve its efficiency by tuning various parameters like execution and optimization threads' time slices and profiling thresholds to achieve earlier injection of optimized translations into execution process while keeping whole system latency acceptable. Besides, IA-32 “halt” instruction can be used as a hint to utilize free cycles and yield processor to optimization thread before the end of execution thread’s time slice. Extensive study of execution and optimization threads' processor time utilization was made in [17].

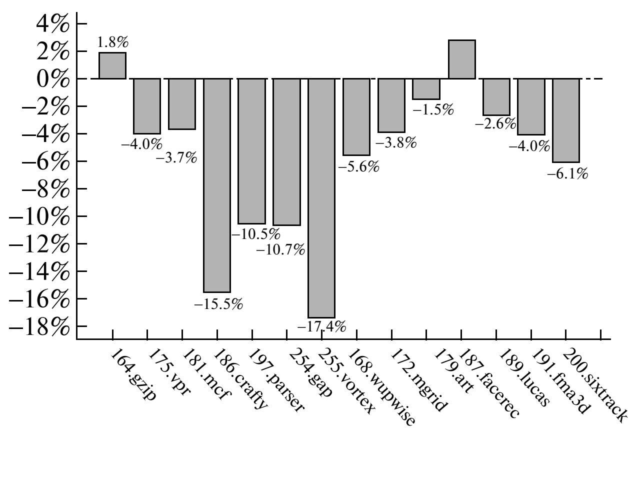

**Fig. 8.** Binary translation slow-down when interleaving optimization with execution (as compared to consecutive optimization).

### 3.3. Background Optimization in a Dual-Core System

In a dual-core system LIntel completely utilizes the second (unemployed otherwise) processor core to perform dynamic optimization in a background thread. In this case execution thread exclusively utilizes its own core and only interrupts execution to acquire next region for optimization and allocate generated translation when optimization completes.

As Fig. 10 demonstrates, overlapping of execution and optimization by moving optimization thread onto a separate core not only eliminates the problem of latency, but also increases overall binary translation system performance.

The resulting speed-up (6% on average) agrees good enough with dynamic optimization overhead estimated for the case of consecutive optimization (see Section 2.1).

### 3.4. Discussion and Future Works

As noted above, selection of hot regions in execution thread gets blocked unless optimization phase completes. However, profile counters continue to grow, and by the end of optimization there may be several nonoverlapping regions in the profile graph with counters exceeding threshold. As counters are checked during execution of corresponding translated codes, next optimizing translation will be launched for the first region executed. Not necessarily will this region be the hottest one. As such, a problem of suboptimal hot region selection arises which also needs to be addressed (profile graph traversal can be quite time-consuming and is not an option).

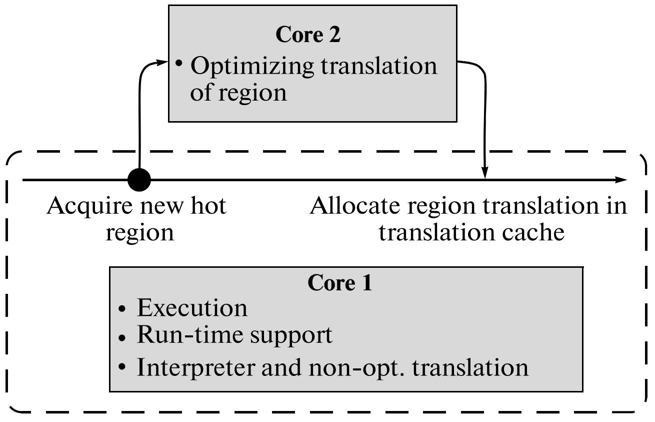
Core 2
• Optimizing translation of region
Acquire new hot region
Allocate region translation in translation cache
Core 1
• Execution
• Run-time support
• Interpreter and non-opt. translation

**Fig. 9.** Utilization of a separate processor core for dynamic optimization.

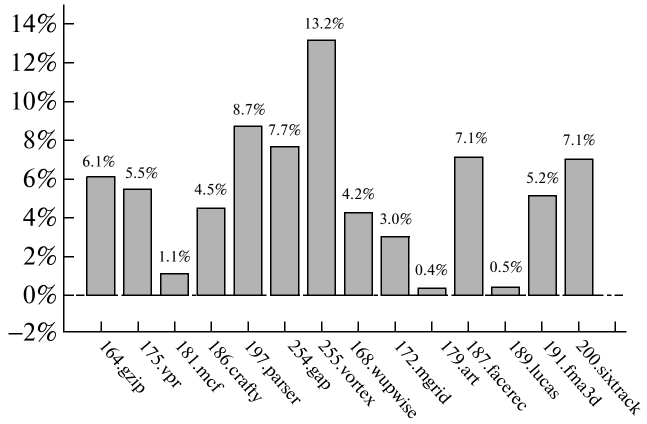

**Fig. 10.** Binary translation speed-up when optimizing on a separate processor core (as compared to consecutive optimization).

The profile of binary translation for SPEC2000 tests (Fig. 4) suggests that current optimization workload is not enough to fully utilize optimization thread affiliated processor core, which will run idle most of the application run time. To improve its utilization ratio, optimizing translator can be forced to activate more often. This can be achieved by dynamically decreasing of hot region profiling threshold depending on current load of the core affiliated with optimizing translator. When execution activity is naturally low, this core should be halted due to energy efficiency reasons.

This is reasonable to ask why not utilize unemployed processor core to execute source binary codes. In other words, if there are more than one target architecture microprocessor core in the system, source architecture system software (e.g. operating system) could "see" and utilize the same number of cores. Current Elbrus architecture implementation (used in this paper) does not satisfy IA-32 architecture requirements concerning organization of multiprocessor systems. As a result, IA-32 multiprocessor support is not possible on top of Elbrus hardware. But we hope to implement this scenario in the future. Still, we believe that having processor cores solely utilized for dynamic optimization is reasonable due to a following:

• different classes of software (legacy software, software for embedded systems, etc.), not always developed with multiprocessing or multithreading in mind, can benefit from multicore or multiprocessor systems when being executed through binary translation with background optimization option;

• keeping in mind the tendency towards ever increasing number of cores per chip, it seems reasonable to utilize some cores to improve dynamic binary translation system performance; not only optimizing translator can consume this resources; other jobs that could also be performed asynchronously include identification and selection of code regions for optimization [18], software code prefetching [19], persistent translated code storage access [20],[3] etc.

Finally, we think that a promising direction for future research and development is building a binary translation infrastructure that could support unrestricted number of execution (in terms of source architecture virtual machine; so that operating system under translation could "see" more than one processor core), optimization and other threads and schedule them efficiently across the available computational resources depending on their quantity, load and binary codes execution behavior.

## 4. CONCLUSIONS

The paper addresses the problem of optimization overhead in dynamic binary translation systems and presents the application of background optimization technique in full system dynamic binary translator LIntel. Implementations for single-core and dual-core systems are considered. In the first case backgrounding is implemented by interleaving execution and optimization, while in the second case dynamic optimization is completely moved onto a separate processor core. In both cases background optimization solves the problem of high latency caused by dynamic optimization which is particularly important for full system execution environment. Performing optimization on a separate core also eliminates optimization overhead from the application run time thus improving binary translation system performance in general.

[3] Asynchronous access to a persistent code storage (aka CodeBase) has already been implemented in LIntel by the moment but is not covered in this paper as we only consider the effect of background optimization implementation.

## REFERENCES

1. Smith, J. and Nair, R., *Virtual Machines: Versatile Platforms for Systems and Processes*, Morgan Kaufmann, 2005.
2. Campanoni, S., Agosta, G., and Reghizzi, S.C., ILDJIT: a Parallel Dynamic Compiler, in *VLSI-SoC'08: Proceedings of the 16th IFIP/IEEE International Conference on Very Large Scale Integration*, 2008, pp. 13–15.
3. Krintz, C.J., Grove, D., Sarkar, V., and Calder, B., Reducing the Overhead of Dynamic Compilation, in *Software: Practice and Experience,* 2001, vol. 31, issue 8, pp. 717–738.
4. Mars, J., Satellite Optimization: The Offloading of Software Dynamic Optimization on Multicore Systems (Poster), in *PLDI '07: 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.
5. Unnikrishnan, P., Kandemir, M., and Li, F., Reducing Dynamic Compilation Overhead by Overlapping Compilation and Execution, in *Proceedings of the 11th South Pacific Design Automation Conference (ASP-DAC '06)*, Piscataway, NJ, USA: IEEE Press, 2006, pp. 929–934.
6. Voss, M.J. and Eigenmann, R., A Framework for Remote Dynamic Program Optimization, in *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, 2000, pp. 32–40.
7. Zhang, W., Calder, B., and Tullsen, D.M., An Event-Driven Multithreaded Dynamic Optimization Framework, in *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT '05)*, USA, DC, Washington: IEEE Computer Society, 2005, pp. 87–98.
8. Guan, H., Liu, B., Li, T., and Liang, A., Multithreaded Optimizing Technique for Dynamic Binary Translator CrossBit, *Computer Science and Software Engineering, International Conference on*, 2008, vol. 5, pp. 945–952.
9. Babayan, B., E2k Technology and Implementation, in *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, UK, London: Springer-Verlag, 2000, pp. 18–21.
10. Volkonskiy, V., Optimizing Compilers for Elbrus-2000 (E2k) Architecture, in *4th Workshop on EPIC Architectures and Compiler Technology,* 2005.
11. Baraz, L., Devor, T., Etzion, O., Goldenberg, S., Skaletsky, A., Wang, Y., and Zemach., Y., IA-32 Execution Layer: a Two-Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-Based Systems, in *MICRO 36: Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, USA, DC, Washington: IEEE Computer Society, 2003, p. 191.
12. Chernoff, A., Herdeg, M., Hookway, R., Reeve, C., Rubin, N., Tye, T., Yadavalli, S.B., and Yates, J., FX!32: A Profile-Directed Binary Translator, *IEEE Micro*, 1998, vol. 18, no. 2, pp. 56–64.
13. Gschwind, M., Altman, E.R., Sathaye, S., Ledak, P., and Appenzeller, D., Dynamic and Transparent Binary Translation, *Computer*, 2000, vol. 33, no. 3, pp. 54–59.
14. Dehnert, J.C., Grant, B.K., Banning, J.P., Johnson, R., Kistler, T., Klaiber, A., and Mattson, J., The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges, in *Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization*, 2003.
15. Klaiber, A., The Technology Behind Crusoe Processors, Transmeta Corporation, Tech. Rep., January 2000.
16. Ermolovich, A.V., Methods of Hardware Assisted Dynamic Binary Translation Systems Performance Improvement, *Ph.D. Dissertation*, Moscow: Institute of Microproccessor Computing Systems, 2003.
17. Kulkarni, P., Arnold, M., and Hind, M., Dynamic Compilation: the Benefits of Early Investing, in *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments, USA, NY, New York: ACM*, 2007, pp. 94–104.
18. Mars, J. and Soffa, M.L., MATS: Multicore Adaptive Trace Selection, in *Proceedings of the 3rd Workshop on Software Tools for MultiCore Systems (STMCS 2008)*, 2008.
19. Mars, J., Williams, D., Upton, D., Ghosh, S., and Hazelwood, K., A Reactive Unobtrusive Prefetcher for Multicore and Manycore Architectures, in *Proceedings of the Workshop on Software and Hardware Challenges of Manycore Platforms (SHCMP)*, 2008.
20. Ermolovich, A.V., CodeBase: Persistent Code Storage for Dynamic Binary Translation System Preformance Improvement, *Inform. Technol.*, 2003, vol. 9, pp. 14–22.