SCIENCE CHINA
Information Sciences

MTCrossBit: A dynamic binary translation system based on multithreaded optimization

HaiBing GUAN[1], RuHui MA[1,*], HongBo YANG[1], YinDong YANG[1], Liang LIU[2] and Ying CHEN[2]

Citation: SCIENCE CHINA Information Sciences **54**, 2064 (2011); doi: 10.1007/s11432-011-4414-5

View online: https://engine.scichina.com/doi/10.1007/s11432-011-4414-5

View Table of Contents: https://engine.scichina.com/publisher/scp/journal/SCIS/54/10

Published by the Science China Press

SCIENCE CHINA
Information Sciences

• RESEARCH PAPERS •

October 2011 Vol. 54 No. 10: 2064–2078
doi: 10.1007/s11432-011-4414-5

MTCrossBit: A dynamic binary translation system based on multithreaded optimization

GUAN HaiBing[1], MA RuHui[1]*, YANG HongBo[1], YANG YinDong[1], LIU Liang[2] & CHEN Ying[2]

[1]*Shanghai Key Laboratory of Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai* 200240*, China;*
[2]*IBM China Research Laboratory, Beijing* 100101*, China*

Received April 16, 2010; accepted January 18, 2011

Abstract Dynamic optimization has been proposed to overcome many limitations caused by static optimization and is widely applied in dynamic binary translation (DBT) to effectively enhance system performance. However, almost all the existing dynamic optimization techniques or methods employed in DBT systems for a single-threaded executive environment considerably increase the complexity of the hardware or the striking runtime overhead. We propose a multithreaded DBT framework with no associated hardware called the MTCrossBit, where a helper thread for building a hot trace is employed to significantly reduce the overhead. In addition, the main thread and helper thread are each assigned to different cores to use the multi-core resources efficiently to attain better performance, two novel methods yet to be implemented in the MTCrossBit are presented: the dual-special-parallel translation caches and the new lock-free threads communication mechanism—assembly language communication (ASLC). We then apply quantitative analysis to prove that MTCrossBit can speed up the original CrossBit. Simultaneously, we present results from the implementation of the MTCrossBit on the uniprocessor machines with multi-cores utilizing the benchmark-SPECint 2000, and illustrate that we achieved some success with the above concurrent architecture.

Keywords dynamic binary translator, hot trace, multithreaded framework, parallelism, multicore

Citation Guan H B, Ma R H, Yang H B, et al. MTCrossBit: A dynamic binary translation system based on multithreaded optimization. Sci China Inf Sci, 2011, 54: 2064–2078, doi: 10.1007/s11432-011-4414-5

1 Introduction

Static complicated optimization techniques, which can make accurate assumptions based on the underlying processor architecture, have been studied for decades. Object-oriented languages and techniques are well used in the modern software domain and have limited the application of static compiler analysis. Because shrink-wrapped software is being shipped as a collection of DLLs rather than a monolithic single-executable, it virtually makes whole-program optimization at static compile-time impossible [1]. Dynamic optimization [2–4] has emerged as an alternative to the above problems. Because a program is dynamically optimized while it runs, optimizations can be specifically customized to the underlying machine architecture, and can automatically adapt to the program's changing behavior at runtime [5]. Currently,

*Corresponding author (email: ruhuima@sjtu.edu.cn)

conventional optimization algorithms, which are widely used in static compilers, including peephole [6], instruction selection [7], graph coloring register allocation [8], cannot be effectively implemented in dynamic binary translation (DBT) systems because they require too much overhead at runtime. That is, not all static optimization algorithms are suited for dynamic executive environments. Because we must perform appropriate optimizations on a DBT system on the fly, the overheads of algorithms themselves also need to be accounted for. If the overhead of an optimization algorithm is more than the benefits that we want to achieve through the optimization algorithm, this method may make a DBT system run even slower. In fact, this is indicated by many classic experiments, such as using peephole as the optimization method applied in a dynamic optimization system [9].

CrossBit [10, 11] is a resource and retarget capable DBT system with intermediate representation [12]. Until recently, it has fully or partially supported guest platforms including SimpleScalar, IA32, MIPS, SPARC, and has fully supported the IA32 host platform. Other RISC instruction set platform hosts are available, for instance, PowerPC and SPARC. Despite the original CrossBit being adopted and optimized by many dynamic optimization methods, for example, in linking [13] and hot trace building [1] techniques, the slowdown is still 3–4 times the native machine codes' execution time. The problem of low performance also exists in some other DBT systems, such as QEMU [14]. However, no matter which conventional optimization method is employed, virtually only one of the operations in existing dynamic optimization systems, such as profiling [15], optimization, and execution interleaving, can be done at a time in a single-threaded environment. Consequently this cannot adequately exploit the multi-core resources in a multi-core environment.

Nowadays, multi-thread optimization techniques can gain performance improvements by assigning multiple threads to run on different CPU cores concurrently [16]. Because optimizations in DBT systems can be concurrently executed with other operations, this technique is suitable for eliminating the limitation mentioned above. Hence, there is also significant enhancement potential for any dynamic systems in the multithreaded environment. This paper proposes a novel dynamic binary translation system based on CrossBit, called Multithreaded CrossBit (MTCrossBit) which is not associated with any special hardware [11]. That is, in parallel with the program's execution, a software helper thread, employed to boost the run-time performance of the original system through some optimization algorithms, is inserted into the CrossBit. It is assigned to another core to utilize a multi-core resource, rather than using the same core occupied by the main thread. This provides an independent executive environment for the main thread that also requires little overhead from the helper thread. We chose the hot trace building optimization algorithm as an experimental mechanism in the helper thread because Baraz et al. [4] have shown that the algorithm of building hot trace is an effective method for enhancing performance. Altering the original execution flows of DBT system frameworks into a multi-threaded one creates some new challenges, such as cache writing/reading collision, the communication between threads and mutual exclusion, and so on. We also introduce several methods to cope with these challenges directly or indirectly in the MTCrossBit. A quantitative analysis is provided to illustrate that this framework is an efficient way to speed up DBT systems at runtime.

Our work differs from previous work in a number of ways, and our main interest is in creating a new multithreaded framework for DBT that is fast enough and systematic enough to be worth using in almost every virtual machine, with the multi-core technique gradually predominating in the processor development domain. To this end, we make the following contributions:

- Performance and multithreaded architecture can be achieved without hardware complexity. This indicates that the multithreaded DBT system can be widely applied in many virtual machines, without accessing specialized hardware. That is, the entire software execution model, with no limitations on hardware as major impediments to extending its application domain, is the key to increasing in a practical way the flexibility, scalability, and compatibility of the systems. It also prevents the customer from spending too much money or time on researching how to produce specialized, compatible hardware to significantly enhance the system performance. Furthermore, the performance of the multi-threaded framework, MTCrossBit, is mainly from the efficient use of the multicore resource because distinct threads are executed on different cores (private cores) to keep them busy.

• Dual-special-parallel translation caches and a novel communication mechanism—ASLC. Some traditional DBT systems, such as, Strata [17] and Walkabout [8], often adopt a single target code cache to store all the translated blocks in the unique code cache, which can be a potential hazard in concurrent system architectures. To solve this problem, dual-special-parallel translation caches, used to store two distinct translated basic blocks, are applied in the MTCrossBit system. Hot trace building optimization is triggered by a hot spot which has been run for more than a specific threshold detected by the profiling instructions. This information is acquired at a machine-language level so that it is difficult to pass these arguments to other threads. Under these circumstances, a novel communication mechanism between threads, called assembly language communication (ASLC), is also presented, which is performed at the assembly language level to avoid extra operations of lock/unlock. Compared with other communication mechanisms (e.g., Producer/Consumer), ASLC also fully avoids the overhead trigged by asynchronous execution between the main and helper threads.

The remainder of this paper is organized as follows. Section 2 reviews related work. Section 3 describes the MTCrossBit architecture, quantitative analysis, and the hot-trace-building algorithm. Section 4 introduces some solutions to the difficult problems encountered along with building the MTCrossBit, such as the lock/unlock algorithm, communication mechanism and special translation caches, etc. Section 5 shows the experimental results of MTCrossBit, and provides a detailed analysis. Finally, we conclude the paper in section 6.

2 Related work

In this section, we will discuss some existing classic dynamic binary systems integrated with conventional optimization techniques or algorithms to achieve high performance. In addition, we analyze briefly how our work relates to similar studies of other systems. Some researchers have focused on special hardware support for better performance, so gradually co-designed dynamic optimization systems are appearing, such as ADORE [18], and Trident [5]. The binary optimization framework ADORE based on a hardware profiling mechanism proposed by Lu et al. [18] focuses on using performance monitoring hardware instead of instrumentation to detect the hot regions and bottlenecks. The Trident is the closest runtime multithreaded framework to our MTCrossBit. It employs hardware support to identify a hot path during the execution of the program, and simultaneously uses spare threads which are called helper threads on a multi-core processor to perform dynamic optimization.

Compared with Trident, MTCrossBit is a dynamic binary translator rather than a dynamic binary optimization tool. The excellent performance incurred by multi-threads in Trident comes basically from heavy-weight optimizations like value specialization, while MTCrossBit uses a simple pure-software helper thread to build hot traces to improve its runtime-performance, without any hardware support. Furthermore, the profiling method used by ADORE and Trident is based on special hardware, so it is strictly limited to special hardware architecture. Because MTCrossBit is designed to be resource and retarget capable and the framework of Trident cannot be applied effectively in MTCrossBit, we employ a more flexible yet effective software instrumentation method to profile the application. In addition, MTCrossBit, in comparison with others, is inherently a multi-core-friendly system, and its special multithreaded framework enables adequate use of the multi-core resources.

3 MTCrossBit design

Binary translation requires machine-level analysis to transform source binary code into target binary code, either by emulating features of the source machine or by identifying such features and transforming them into equivalent target machine features. CrossBit, as a dynamic binary translator, designed with resource and retarget capability in mind, is classified in the latter type of transformation. During execution in the main thread, the performance of the prototype system was enhanced significantly through optimization concurrently executing in the helper thread. This multicore-friendly framework, called MTCrossBit, will

be explained in detail in this section.

3.1 Motivation and discussion

Advances in computer science and technology are continuing to drive Moore's Law and double the number of transistors available on a chip every two years [19]. How to use these transistors efficiently becomes a challenge. Indeed, out-of-order execution and much speculation give these monolithic chips another opportunity to achieve performance improvement. However, many issues have arisen to prevent large, monolithic uniprocessors from continuing as a viable design, leading the way to the rapid emergence of multicore processors [20, 21]. Multiprocessors integrate several cores across the chip instead, and are driven by Moore's Law. Although multicore processors are available, traditional software, such as OS, VMM, compilers and other applications cannot adapt well. Consequently, researchers and designers seek to discover new methods to utilize multicore resources effectively. Thus, extracting greater concurrency to keep all or most of the cores busy for multiprocessors is a method available for OS/VMM to achieve more performance. In virtual machine research, many tradition DBT systems benefit from this method. From section 2, we can see that performance improvement is achieved through adding special hardware or complex heavy-weight software. These limit the scalability of multicore platforms, without considering their further development. In this study, on a fundamental research platform running on a multiprocessor physical machine, the MTCrossBit is constructed by inserting another software thread, which runs on some parallel-available parts corresponding to the main thread. It not only achieves improved performance, but also gives researchers an opportunity to do further research on the multicore platform.

Note that the extra overhead triggered by adding the helper thread can be negligible. Traditionally, the overhead between threads is mainly from frequent context switches on a uniprocessor platform. Fortunately, in this multithreaded framework, MTCrossBit, the on-going overhead caused by redundant context switches competing for CPU resource cannot be triggered. Indeed, the unique overhead caused by the helper thread is a communication mechanism between threads. If the traditional communication mechanism between threads, such as Producer/Consumer, is employed, this leads to an extra overhead (this is discussed in subsection 4.3). To avoid this overhead, another new lock-free communication mechanism, ASLC, is presented, including the Producer, Critical section, and Consumer. Note that this mechanism only causes a small write operation overhead, that is, this overhead is merely caused by the Producer. The critical section affected is only a section of memory space, while Consumer is deemed a read operation and cannot create any extra overhead because it is executed in parallel with the main thread. In addition, the write operation consists of only a few instructions and cannot create too much overhead. That is to say, the advantages of adding the helper thread far outweigh the disadvantages.

3.2 Overview

CrossBit is a resource and retarget capable dynamic binary translation system, as well as a process virtual machine designed mainly to provide a platform independent computing service for a new virtualized executive environment. As the multi-core technology gradually becomes even more prevalent, multithreading techniques will be a significant key to making adequate use of more resources corresponding to multi-core processors. To achieve both high quality executable basic blocks and a lower overhead during optimization, we propose a multithreaded dynamic binary translation framework, MTCrossBit, to further improve performance with chip multiprocessors. As shown in Figure 1, we see the framework of MTCrossBit as a multi-threaded dynamic binary translation system, rather than a dynamic optimization system. The Image Loader is responsible for analyzing the guest executable file and loading its code and data into the memory space, that is, the guest's memory image. The Image Loader is also considered to be a bootstrapper to initialize the system. Indeed, the identification of the executable nature of the file, the mapping of every executable section, the loading of the runtime load, and some dynamic linkage library issues, are dealt with by the Image Loader. The execution engine then transfers the control to the translated basic block cache (TCache) manager which looks up the translated basic blocks in the Basic

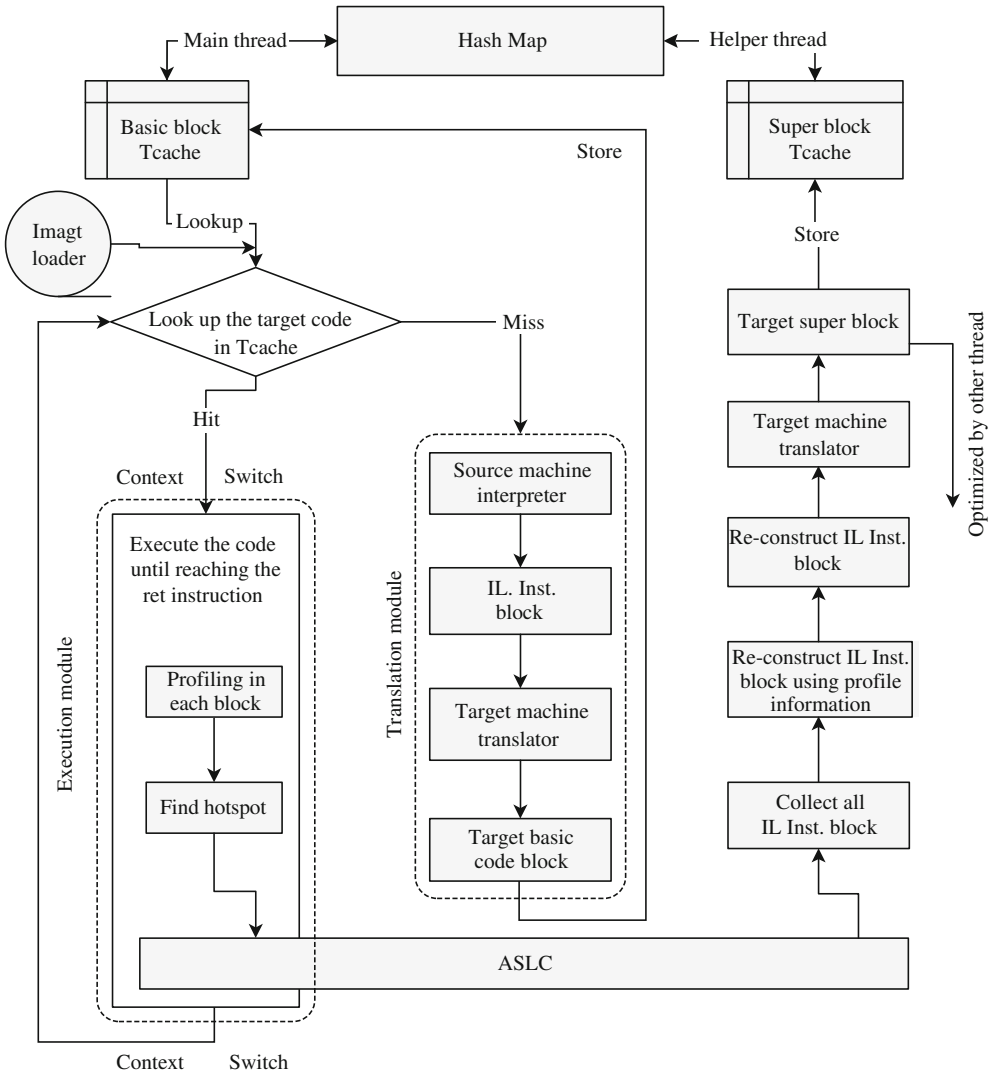

Figure 1 MTCrossBit framework.

Block TCache (it stores many unoptimized translated basic blocks in the main thread) using the source program counter (SPC) stored in the Hash Map. In fact, this look-up operation determines whether a special translated basic block exists in the Basic Block TCache, corresponding to the new basic block before the operation of translation.

• If the special translated basic block is not cached (called a cache miss), this would lead to a new translation process executed by the translation module described in Figure 1. This may happen when the required translated basic block is not in the TCache or is evicted by the TCache manager. In the entire translation process, the key point to successful translation is the intermediate representation, also called the intermediate language instruction (IL Inst), which easily consolidates various instruction sets into the source and target machines. In detail, based on the intermediate representation, the translation process is clearly divided into two phases: interpreting and translating, both of which are time-consuming. In the interpreting phase, the engine executes the binary decoding of the source code in the basic block and accomplishes the operation of binary transformation to the intermediate representation. On the other hand, the translating phase performs encoding, transforming the binary representation to a target basic block (translated basic block) that can be compatible with the target machine, and registering allocation. When the translated basic block is completed after the interpreting and translating phase, it is stored in the Basic Block TCache, and the engine updates the Target Program Counter (TPC), as a pointer as well as the SPC, into the Hash Map.

• If successful, this operation returns the translated basic block's entry address for execution, that is, the execution module is reactive. In this process, a context switch would happen twice: the first context switch to the TPC occurs when the engine needs the translated basic block's entry address after successful lookup by the SPC because the relationship between the SPC and TPC is one to one mapping in the Hash Map. After executing the translated basic block, the latter context switch to the SPC is triggered by another application to the look-up operation performed by the SPC. Note that the profile operation for each block to achieve hot spot information happens in this phase, which is the key point to constructing a hot trace in helper thread.

When the DBT system is executing the translated basic block found by the SPC in the Basic Block TCache, we can directly collect the profile information from each basic block, for example context information, linking information, the execution counter of the basic block, and other runtime information. The accuracy of runtime profiling often determines the ultimate performance of the dynamic optimization, and the information profiled is not only used to determine the hot trace and guide the generation of superblocks, but also provides MTCrossBit with sufficient information for other instrumentation work. After collecting each block's profile information, the engine detects the hot spot accordingly. Although hot spots are detected with ease, the work to pass this information to another thread is difficult for the traditional communication mechanism to do with a lower overhead. In this study, a novel communication mechanism, ASLC, is capable of tackling the complex communication between threads successfully, and is described in subsection 4.3. Consequently, because the hot spot is the translated basic block associated with the larger number of the execution counter or avails itself of the other basic blocks, the engine will mainly collect all the IL Inst blocks using the hot spot information (profile information) to reconstruct the IL Inst B block-like translation in the main thread. As explained later, in the superblock thread, the binary representations will be transformed into the superblock (target basic block), which is composed of several translated basic blocks. In addition, the engine will optimize the superblock with a more aggressive algorithm in another thread (this will be dealt with in a future study). In the end, the superblock is stored in the Super Block TCache, and the system updates this superblock into the Hash Map through the SPC. The main thread can execute this superblock (the optimized hot trace) also by looking up the Hash Map.

In the MTCrossBit, each thread has its own cache to avoid concurrent cache operation collisions, while two code caches are employed to store the various blocks. The availability of all these caches is controlled by a single Hash Map. The read operation of the Hash Map can be implemented between concurrent threads without any conflict and the update into the Hash Map is in sequence, avoiding any write hazards.

3.3 Quantitative analysis

Optimizations realized by the multithreaded method we propose are better than the traditional serial dynamic optimization methods in a dynamic binary translation system or in a dynamic optimization system. To verify this conclusion, a quantitative example is given here. Therefore, we define an operator $Time(M)$ which represents the time consumed in executing module M once. In a real executable file F, without loss of general applicability, we evaluate pieces of binary instructions that are simply constituted using three binary basic blocks: B_1, B_2 and B_3. All the conditions follow:

• Suppose that the time spent executing each of the basic blocks B_1, B_2 and B_3 once is all equal to 10 milliseconds, where B_1 runs for 20 times, B_2 runs for 40 times, and B_3 runs for 1000 times.

• A hot spot here means a basic block that has run more than 50 times, so that we can only have basic block B_3 for hot trace building.

• If there is a compelling optimization algorithm, optimization algorithm that can be performed on any basic block B_i, the associated overhead that we denote as $OverHead(B_i)$ is 80 times $Time(B_i)$ (800 ms). After B_i is optimized as block ΔB_i, the $Time(\Delta B_i)$ is equal to $0.8 \times Time(B_i)$ (a 20% speedup).

We then adopt the following three models to execute file F:

(1) Executed in CrossBit but without any optimization. After loading the executable file F, it continues to perform interpreting, translating and executing in sequence (a miss in the TCache) or only executing directly (a hit in the TCache). Without considering whether it hits or misses in the TCache, the total

time for running the non-optimization executable file F is computed by

$$Total_F = \sum_{i=1}^{3} Time(B_i) \times N_i. \tag{1}$$

Indeed, like a conventional complier proceeding, this execution model in series is not effective in performing the application due to non-optimization.

(2) Executed in CrossBit but with dynamic optimization. Along with the optimization algorithm being performed dynamically on the hot spot B_3, the overhead for the optimization must be included in the total execution time, in comparison with static optimization. Considering that B_3 will not be viewed as a hot spot until it has run 50 times, we calculate it through

$$Total_F = \sum_{i=1}^{2} Time(B_i) \times N_i + OverHead(B_3) + Time(B_3) \times 50 + Time(\Delta B_3) \times (N_3 - 50). \tag{2}$$

Optimizations that are based on static analysis in traditional compiler technology can enhance the performance of the program, but often incur a heavy cost at runtime. In the dynamic execution environment, the choice of optimizations to be performed significantly affects the balance between the time spent outside program execution and the effectiveness of the data collection process. It is important to ensure that the benefits of applying dynamic optimization outweigh its cost. If when executing in this way the execution and optimization are sequential rather than in parallel, that is, executing in a single-thread environment, the result of this framework is inadequate.

(3) Executed in MTCrossBit but with parallel optimization. Using the dynamic multithreaded optimization mechanism means that when the profiling information comes to a threshold during execution, the executable file will continue to run the un-optimized previous basic blocks while the optimization thread works in parallel. Because the execution times of B_3 take a large proportion, we only care about how many times B_3 is executed before and after creating ΔB_3. Thus, we calculate the total time by

$$\begin{aligned} Total_F \ = & \sum_{i=1}^{2} Time(B_i) \times N_i + Time(B_3) \times \left(\frac{OverHead(B_3)}{Time(B_3) + \Sigma(Time(B_n))} + 50 \right) \\ & + Time(\Delta B_3) \times \left(N_3 - 50 - \frac{OverHead(B_3)}{Time(B_3) + \Sigma(Time(B_n))} \right). \end{aligned} \tag{3}$$

From formula (3), we can see that the execution time of B_3 consists of two parts. When the number of execution times of B_3 is less than 50, the helper thread for B_3 is idle. Once B_3 is executed too frequently (the number of execution times is greater than 50), the helper thread is triggered to perform optimization. Note that B_3 may be sequentially executed in the main thread, or if the main thread is occupied by other basic blocks (one or more), the execution time of which can be represented as $\Sigma(Time(B_n))$, the helper thread is then triggered. The worst situation is when B_3 is performed on the main thread the whole time until the optimization algorithm finishes its optimization work. Based on this, $Total_F$ can be represented as

$$\begin{aligned} Total_F \ = & \sum_{i=1}^{2} Time(B_i) \times N_i + Time(B_3) \times \left(\frac{OverHead(B_3)}{Time(B_3)} + 50 \right) \\ & + Time(\Delta B_3) \times \left(N_3 - 50 - \frac{OverHead(B_3)}{Time(B_3)} \right). \end{aligned} \tag{4}$$

Figure 2 shows the total overhead of this executable file F with these different optimization mechanisms. Through comparison, we can see that multithreaded optimization mechanism applied on a hot trace is obviously an effective way to improve the dynamic system's runtime performance. Assessing the dynamic optimization with and without multithreading, the speedup improved with the multithreaded technique in our example is

$$((600 + 8100 + 800) - (600 + 8260)) \div (600 + 8100 + 800) \times 100\% = 6.74\%.$$

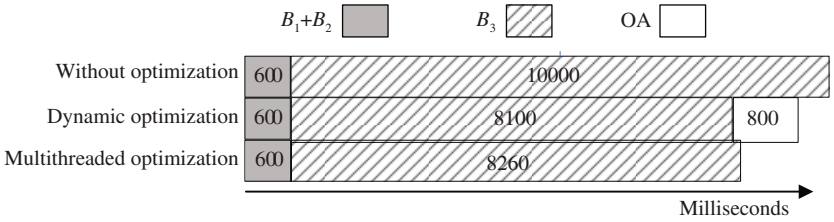

Figure 2 Execution time for the file F with different optimization mechanisms.

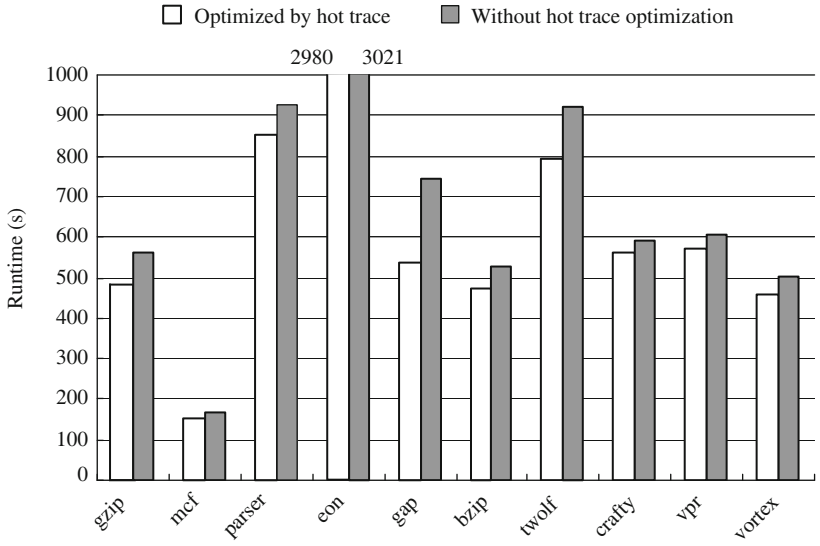

Figure 3 The performance of traditional CrossBit with and without hot trace optimization.

Without considering the real execution environment, the number of blocks like B_1 and B_2 (executed a few times) and the hot traces like B_3 (executed many times) would be significantly large during execution. Furthermore, B_3 cannot be executed on the main thread while the optimization algorithm is doing its work. Finally, as the optimization is performed in parallel, it is apparent that the more optimizations we adopt, the higher the achievable speedup.

3.4 Hot trace and superblock

It is often known that a program spends 90% (or more) of its execution time on 10% (or less) of its code. A common technique for reducing overhead is to classify the dynamic instruction streams into frequently-executed (hot) streams and infrequently-executed (cold) streams. That is, sections of program code that are found to be executing very frequently are candidates for optimization. In this study, we selected the method for building a hot trace as the optimization algorithm mainly in the conventional CrossBit, the dynamic binary translation system, and through experiments on SPECint 2000. The results of the traditional CrossBit with and without hot trace optimization are shown in Figure 3.

Figure 3 shows how important the original CrossBit is to hot trace optimization. In addition, the change in data is displayed in the above histogram, which is expressed as the greatest disparity between the white column and grey column in every benchmark group, and the most significant changes in the gap benchmark ranged from 742 to 535. In virtually all cases, the performance improvement was more or less enhanced by building a hot trace. However, while building a hot trace reduces extra costs, in sequential execution, some overheads can be avoided through being executed in parallel, including code reconfiguration, optimization between blocks, and the elimination of dead code.

In MTCrossBit, the hot trace optimization strategy applied in the helper thread is performed concurrently with the system execution in the main thread. In contrast to CrossBit, the goal of this multithreaded framework is to lower the overheads in hot trace building and in the overheads of context

switches between the optimized and un-optimized blocks during execution. Meanwhile, the selection of a hot trace algorithm as the system's main optimization policy in MTCrossBit, is also based on the following three primary factors:

1. Hot trace algorithm is capable of architecture parallelism. The reason we selected the hot trace algorithm as the optimization method in the helper thread is not only the significant performance achieved, but also its independent architecture. Indeed, the process of building one hot trace is that some profiled information about the translated basic blocks will provide the DBT system with many hot spots. The phase of hot trace building is triggered depending on the hot spots information passed into the helper thread from the main thread through the ASLC mechanism. In the traditional CrossBit, if the system wants to construct a hot trace, other operations (e.g. translation, execution and profiling) are fully suspended and extra context switches must be protected. However, constructing one hot trace is independent of other operations in MTCrossBit. In other words, profile operation, basic block translation, optimization, or execution can be concurrently executed with the hot trace building in different threads. This provides an opportunity to enhance performance.

2. It is comparatively easy to realize. Because CrossBit implements its optimization in a function call manner, if we can use the multithreaded programming technique to further reduce the overhead of building the hot trace or context switch, we can also apply this method to some other optimization algorithms.

3. Aggressive algorithms with high time complexity should be performed on the hot traces. As we know, the execution time consumed in optimizing has to be less than the dynamic execution time in a DBT system. While the program exits, the action of optimization has to stop, so we must make sure that all the hottest traces in the binary images have first been optimized. To achieve this goal, all the optimization threads should first deal with all the hot traces, for they have the higher priority. Therefore, a multithreaded hot trace building framework can form the backbone for other optimizations in our future work.

Apparently, it is crucial for the system optimized by the hot trace to be able to determine whether a block is hot. In the helper thread of MTCrossBit, a threshold value as a constant defined by the programmer is inserted into the system as a consequence of the results of several significant and typical experiments. In addition, in this study, we choose an execution time threshold of 3000 to judge a hot spot, which has been established by our experiments.

When a hot spot is validated, then the engine collects all the IL Inst associated with the hot spot and reconstructs the IL Inst block to be translated. The translated basic block after reconstructing the IL Inst block is called a superblock, which usually has several basic blocks related to the basic blocks translated in the main thread. To achieve better performance, linking as an efficient optimization method has been widely utilized to chain basic blocks and superblocks. In CrossBit, the basic translated blocks and superblocks are together cached in an exclusive cache-TCache. However, in MTCrossBit, they are cached in Basic Blocks TCache and Super blocks TCache, respectively. Though this system can completely avoid interlacing a cache, linking two distinct caches also brings the problem of mutual exclusion.

4 Several challenges

Along with constructing the MTCrossBit, several intractable but necessary-solved problems are encountered, including mutual exclusion, communications and code access among threads. Despite there being no easy solutions to these obstacles, we have still found some effective solutions.

4.1 Linking

Linking is an optimizing method performed on all the basic blocks by modifying the machine codes after they are executed once [13]. All the hot traces also need to be linked after they are created. Nevertheless, in MTCrossBit, when linking, the recently built superblock should not be executed immediately because linking would modify it at the same time. That is to say, hot trace linking is mutually exclusive to the

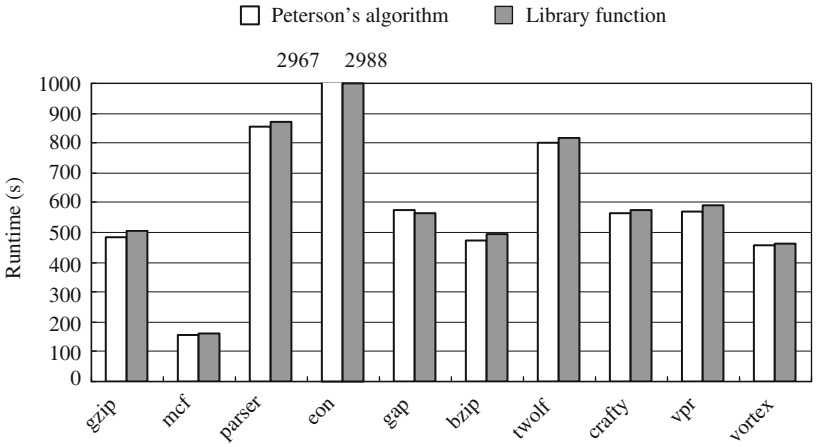

Figure 4 Performances of various lock/unlock algorithms investigated using SPECint 2000.

main thread's executing module. So not only do we avoid modifying the target-machine codes being executed simultaneously, but we also acquire fast superblock linking from the helper thread to the main thread with an adaptable lock/unlock algorithm, which can transform these two threads in a critical section. A translated basic block that is executed by only one thread at a time is employed in the MTCrossBit to solve this problem better.

Peterson's algorithm, as a lock/unlock algorithm, is a concurrent programming algorithm for mutual exclusion that allows two threads to share the same resource without conflict, using only shared memory for communication. As the simplest known solution, it is employed in MTCrossBit to resolve the two-thread mutual exclusion problem on a platform that only provides read-write atomicity. Correspondingly, to assess the impact of this algorithm in MTCrossBit, we apply Peterson's algorithm and the *pthread* function (it is a library function to solve mutual exclusion from the Glibc library in Linux OS) on a dual-core processor machine and the results are shown in Figure 4.

Figure 4 shows the double lock/unlock algorithms, Peterson's algorithm and the Library function, being applied to our multithreaded DBT system, MTCrossBit, to settle potential mutual exclusion between threads so that further execution improvements would be readily acquired. Clearly, the execution time of Peterson's algorithm spent on the SPECint 2000 benchmarks is far less than that for the Library function, except for the gap benchmark. Regardless of the eon benchmark which has been used too often to be ignored when researching the performance time of the entire system, on average, the time spent on the SPECint 2000 benchmarks is merely 555.7 s using Peterson's algorithm, while even the mcf benchmark takes only 157 s. When the system uses Peterson's algorithm, the total runtime decrease relative to the Library function is 70 s. Although the performance time of the eon benchmark in MTCrossBit associated with the lock/unlock algorithm (either Peterson's algorithm or Library function) is significantly the longest compared with others, the enhancement in speedup by the application of Peterson's algorithm is higher than the others. Because the benchmark eon has many float computations itself (the system of MTCrossBit is not perfectly suited to excessive float computations) and the length of its basic block is longer than other benchmarks, it is an understandable and anticipated fact that we obtain a far longer execution time.

Meanwhile, the increasing execution time for the gap ranges from 585 to 634 s after performing the Peterson's algorithm because this benchmark uses too many parts to test big or complex codes, which does not fully adapt to MTCrossBit. This change, which does not overly impact on the entire system's performance, differs from others due to the special architecture of MTCrossBit and we intend to research this dilemma as soon as possible. Although a few flaws exist, the performance improvements would be significant, consequent to the insertion of Peterson's algorithm in MTCrossBit. Despite the Library function, the *pthread* function has dominated and been applied to many programs, although it cannot be adapted to the multithreaded framework, MTCrossBit (it only has two threads currently), with better performance. Overall, we selected the Peterson's algorithm as the lock/unlock algorithm in MTCrossBit

due to its dominant execution performance, rather than the Library function or the conventional binary semaphore.

4.2 Dual-special-parallel translation caches

TCache employed to cache both translated basic blocks and superblocks has been used widely in the DBT system. There are two excellent features of TCache that are also the reasons why so many DBT systems select it to cache translated blocks. First, it is controlled easily because it is only a software cache, a sequential address space in memory, rather than a physical cache. On the other hand, the information on the cached blocks need not be exchanged with other sections of memory, that is, the architecture of this information is not characteristically hierarchical. Because a translated block inserted into TCache is to be executed through being transferred or replaced, the writing policy in TCache need not to be considered, while this must be seriously considered in a physical cache. Indeed, not only CrossBit but also MTCrossBit also choose TCache to achieve even fewer translation counters and context switches.

In CrossBit, the translated basic blocks and superblocks are all inserted into TCache to be reused to decrease the translation counters. However, if a unique translation cache is contained in the prototype system, in MTCrossBit as well as CrossBit, collision occurs when reading/writing in the translation cache. That is, when a code block is only translated but not optimized in the main thread, and a superblock after rebuilding a hot trace in the helper thread would be concurrently stored in a unique translation cache. This may cause an unpredicted data or control collision by competing for the same address. To avoid this serious problem, two methods exist:

1. Lock/unlock algorithm. Mutual exclusion also frequently happens when several writing operations are synchronously executed on the unique code cache. That is, when two distinct writing operations, respectively controlled by the main thread and helper thread, are simultaneously enacted on the unique cache, TCache, mutual exclusion will be triggered because only one writing operation is permitted to be executed in TCache. Encountering this serious problem, we choose a lock/unlock algorithm to solve it. The algorithm is applied to the TCache, which is deemed the critical section. Although the lock algorithm is considered a good solution, once it is reactive, another thread will be immediately suspended. That is, the suspended thread cannot be executed until the TCache unlocks. This process brings extra unnecessary overhead, such as the waiting time of the suspended thread, the waiting time of the system's sequential execution, and other optimizations in TCache.

2. Dual-special-parallel translation caches. The method, as mentioned above, causes the execution between threads to be asynchronous, that is, waiting time exists between threads. This overhead is an extra overhead, the by-product of the lock/unlock algorithm, which can be completely avoided. Consequently we propose a lock-free and no waiting-time method, dual-special-parallel translation caches, which can better solve the mutual exclusion in the TCache. It is used to store two distinct translated blocks (translated basic blocks and superblocks). Note that it is not merely that the result comes from dividing the unique TCache into two special caches, and each cache is a thread-private cache, but also that the objects stored in the dual caches can communicate with each other through linking. Along with writing basic blocks into the respective translation caches, the operation of reading/writing with translation caches is controlled by the hash function. The detailed mapping relationship of the hash function is determined by the last four bits of the hexadecimal entry address which is the key to acquiring the corresponding mapping result. For instance, assuming that the entry address is denoted by 0x4005678, the offset of it is simultaneously 5678. The data, 5678, as the exclusive memory addressing data, indirectly indicate that the simplest addressing mode further consolidates the memory management without extra cost.

In MTCrossBit, we define sequential memory spaces, called a translation cache (its space size is 10M), as being assigned to access the translated basic blocks. This method is also applied to define another translation cache. We note that the dual translation caches taking on the ability of being readable/writable/executable are triggered by the library functions *mmap*() and *malloc*() in the Linux OS.

4.3 Lock-free communication mechanism

As we can see in Figure 1, the communication happens when the profiling module is going to pass information on hot spots to the hot trace building thread. It is well known that an adaptable and

effective communication mechanism is a key aspect of passing important information on hot spots between threads. However, traditional thread communication mechanisms, such as Producer/Consumer, do not fit MTCrossBit effectively due to their drawbacks.

Producer/Consumer. This model [22] makes all threads operate their critical sections, which employ extra semaphore operations such as SemSend () or SemWait (). The critical section refers to the portion of the program that uses a single non-sharable global resource which is accessed by two or more threads. All these operations need to be performed at the same language level. In fact, Producer/Consumer needs a lock/unlock algorithm to prevent mutual exclusion in the critical section, which is associated with the extra overhead. Otherwise, in MTCrossBit, the profiling module in Figure 1 is realized by the assembly instructions in each basic block while the hot trace building thread is implemented by C++, so we must introduce some extra overheads if we are still to employ this model. This disadvantage comes from the redundant context switching operation, which is used to exit the back-end execution process to perform semaphore operations like SemWait (). The reason context switch occurs frequently in MTCrossBit is that once the profile module finds a hot spot, the context switch is triggered to go back to a high level language to complete passing the hot spot information. In addition, the critical section of this model, a section of memory only allows the access for a specific type of operation. Because in this critical section, only two types of operation, write/read, exist, if the write operation (Producer) is being executed, the read operation (Consumer) must wait to be executed until the other execution is complete (or vice versa). Note that the waiting time cannot be fully ignored.

To make our architecture more effective and sound, a new communication mechanism was proposed:

ASLC(*assembly language communication*). The helper thread first reads the value of the ProducerCount, which is compared with the value of the ConsumerCount (reading semaphore). If the comparison result is unequal, this indicates that the ProducerCount is modified by the main thread because their initial values are 0. Because the helper thread is built in a high-level language, C++, the first address of the successive memory space is easily acquired by the addressing mode used in the high-level language. When the work of building a superblock is finished, the hash map is updated. On the other hand, when the comparison result is equal, the helper thread will continuously wait until the ProducerCount is altered. Compared with the Producer/Consumer communication method, ASLC does not have to suspend the target-machine codes' process of executing the main thread to make any semaphore operations.

5 Performance evaluation

To gauge the performance impact of the helper thread used for optimization in MTCrossBit and to evaluate the relevant tradeoffs, we have applied this virtual machine based on helper threading optimization on modern computers through the test set SPECint 2000 [23]. Not all applications are suited for customized hardware. In this situation, an increasingly popular way to provide performance is through a multi-core technique. As multi-core techniques gradually move to the commercial market from the research domain, the processors with single cores are out of date due to their lower performance when implementing some large applications. In fact, making use of multi-core resources adequately has been the research focus for many researchers. Increased parallelism is one of the effective methods of resolving this issue. Consequently, we implemented our multithreaded framework on a dual-core machine to validate the parallelism of our multithreaded framework.

To achieve greater benefits for MTCrossBit, we chose the latest Linux kernel version and an experimental platform with a 2.00 GHz dual core Intel Xeon CPU and 4 GB memory. All the benchmarks are from SPECint 2000, for which there are only integer benchmarks because of their inherent control flow complexity. We select the CrossBit version with the source-machine architecture of MIPS while the target-machine architecture is IA32 as the MTCrossBit execution version. All the cache sizes (the size is 10M in memory) are set to be large enough for the target code to avoid some blocks being repeatedly translated.

As the hot trace has not been optimized by the third thread mentioned in Figure 1, the speedup invoked by the multithreaded programming technique is currently not very high. Figure 5 shows the

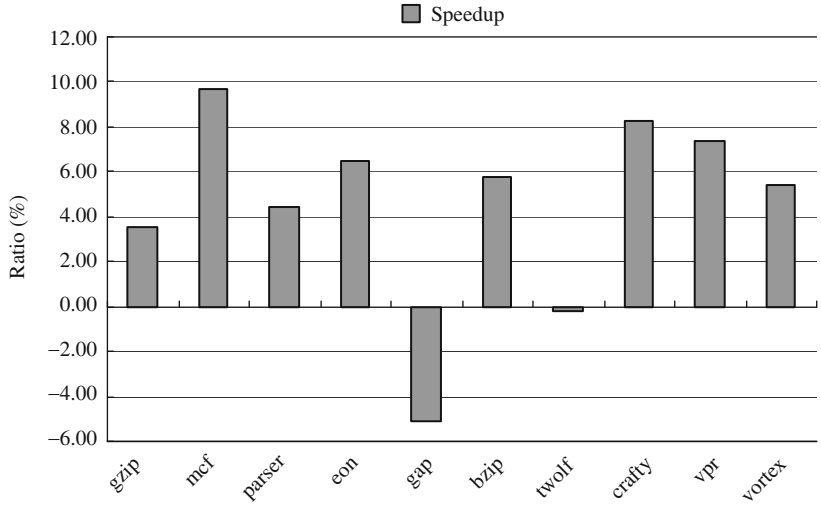

Figure 5 Speedup rates of MTCrossBit and CrossBit with hot trace optimization.

speedup of MTCrossBit compared with CrossBit with the hot trace building without multithreading but in sequential mode (the function call approach). In Figure 5, taking the benchmark *mcf* for example, CrossBit with the function call implementation of hot trace building in sequential mode can run *mcf* for 156 s while MTCrossBit runs mcf in 140 s because the benchmark *mcf* has a simple structure. The core of the benchmark 181.*mcf* is the network simplex code "MCF Version 1.2—A network simplex implementation", which is a specialized version of the well-known simplex algorithm for network flow problems. The linear algebra of the general algorithm is replaced by simple network operations such as finding cycles or modifying spanning trees that can be performed very quickly. The main work of this network simplex implementation is pointer and integer arithmetic. We can see that the architecture of *mcf* is mainly composed of pointer and integer variables to find more loops or modify spanning trees. The multithreaded framework, MTCrossBit, can just cope with more loops or data to build the hot trace.

Clearly, the benchmarks *gap* and *twolf* bring down the performance of the entire system due to their special program architecture. The benchmark *gap* has multiple parts: one of which is the standard *gap* speed-benchmark, exercising mostly the combinatorial functions and big number library, then some test functions for the finite field, permutation group and subgroup lattice computations, a program comparing two different methods of finding normalizers in solvable groups and finally a test exercising the collector for so-called ag-groups. This is a piece where the bulk of the computation is not done by the interpreter. The benchmark *gap* is designed mostly for computing in groups, that is, it is used to test mostly large or complex codes. On the other hand, the benchmark *twolf* is used in the process of creating the lithography artwork needed for the production of microchips. Specifically, it determines the placement and global connections for groups of transistors (known as standard cells) which constitute the microchip. The input files consist of the block description file, the netlist file, the net weighting file and the parameter file. This benchmark is employed mainly to test circuits, which is also one of MTCrossBit's vulnerabilities. Indeed, in MTCrossBit, if the executing process has many large basic blocks or special basic blocks (less loops) to run without encountering the ret instructions, which are used to suspend the executing of target-machine codes, the hot trace linking module cannot access its critical section immediately after the hot trace is built. This may cause the hot traces linked into other blocks to take longer than the function-call approach. This fact has weakened the improvement brought by hot trace building as well as multi-threading, such as in *gap* and *twolf* in Figure 5 (The basic blocks used to constitute them are too long or special). This is why the change in these two benchmarks differs from others.

Although this research problem currently cannot be solved perfectly, it does not affect the eventual result. At the same time, the results mainly illustrate that our work on the multithreaded framework is effective and can be used as a base to continue our research on the optimization thread. This 4.57%

improvement, on average, comes from the overhead reduction in context switching and hot trace building. The conventional CrossBit executed on the dual-core or multi-core machine still behaves as in a single-core machine, for it only possesses the single thread that cannot completely utilize a multi-core resource. As the result from the two architectures executed on multi-core machines, we can conclude that this multithreaded framework, MTCrossBit, can use multi-core resources, and has better compatibility and adaptability with multi-core systems.

6 Conclusions and future work

With multi-core processors, providing efficient performance improvements through increased parallelism is becoming more prevalent. Through a general purpose design which not only attains significant performance improvements but also major reductions in power consumption, a pure-software helper thread is inserted into the original DBT system, CrossBit. This is carried out with no interruption or impact on the execution in the main thread, bringing considerable and significant performance improvements. In MTCrossBit, a conventional optimization method, building a hot trace that is concurrently executed in the helper thread effectively boosts the performance, rather than being executed sequentially in a single thread executive environment. However, when constructing a multithreaded framework based on CrossBit, several difficult but unavoidable problems are encountered, such as mutual exclusion, the access of translated basic blocks, and the communication mechanism between threads. Responding to these problems, several methods, including Peterson's algorithm, ASLC mechanism and Dual-special-parallel translation caches, are proposed as solutions in this study. We applied the MTCrossBit system on a multi-core processor. The results illustrate that the multithreaded framework achieves good progress in a multi-core environment and is also adaptable to fast-development techniques.

With regard to the architecture, the performance improvement based on the MTCrossBit system is significantly enhanced, but the advantage potential for the current MTCrossBit is even larger because in the helper thread no optimization for the hot trace is done at runtime. Therefore, in future work, an aggressive optimization thread (third thread) which is shown in Figure 1 will be added to further optimize the hot trace in the helper thread (second thread). Through this optimization, future multithreaded frameworks should bring greater performance improvements along with the fast development of computer science.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (Grant Nos. 60970108, 60970107), the Science and Technology Commission of Shanghai Municipality (Grant Nos. 09510701600, 10DZ1500200, 10511500102), IBM SUR Funding and IBM Research-China JP Funding.

References

1 Bala V, Duesterwald E, Banerjia S. Dynamo: A transparent dynamic optimization system. In: Proc ACM SIGPLAN Conf on Programming Language Design and Implementation. Vancouver, British Columbia, Canada, 2000
2 Shankar A, Sastry S, Bodik R, et al. Runtime specialization with optimistic heap analysis. In: Proc ACM Conf on Object-Oriented Programming Systems, Languages and Applications. San Diego, CA, USA, 2005
3 Rosner R, Almog Y, Moffie M, et al. Power awareness through selective dynamically optimized traces. In: Proc Int Symp on Computer Architecture. Munchen, Germany, 2004
4 Baraz L, Devor T, Etzion O, et al. IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium(R)-based systems. In: Proc Int Symp on Microarchitecture. San Diego, CA, USA, 2003
5 Zhang W F, Brad C, Tullsen D M. An event-driven multithreaded dynamic optimization framework. In: Proc Int Conf on Parallel Architectures and Compilation Techniques. Saint Louis, MO, USA, 2005
6 Sorav B, Alex A. Automatic generation of peephole superoptimizers. In: Proc Int Conf on Architectural Support for Programming Languages and Operating Systems. San Jose, CA, USA, 2006
7 Pozzi L, Atasu K, Ienne P. Exact and approximate algorithms for the extension of embedded processor instruction sets.

IEEE Trans Comput Aid D, 2006, 25: 1209–1229
8 Lupo C, Wilken K D. Post register allocation spill code optimization. In: Proc Int Symp on Code Generation and Optimization. Manhattan, New York, USA, 2006
9 Sorav B, Alex A. Binary translation using peephole superoptimizers. In: Proc USENIX Symp on Operating Systems Design and Implementation, San Diego, CA, USA, 2008
10 Source codes and Introduction of CrossBit, http://sourceforge.net/projects/crossbit/
11 Li X L, Zheng D E, Ma R H, MTCrossBit: A dynamic binary translation system using multithreaded optimization framework. In: Proc Int Conf on Algorithms and Architectures for Parallel Processing. Taipei, Taiwan, China, 2009
12 Shi H H, Wang Y, Guan H B, et al. An intermediate language level optimization framework for dynamic binary translation. ACM SIGPLAN Notices, 2007, 42: 3–9
13 Hiser J D, Williams D, Hu W, et al. Evaluating indirect branch handling mechanisms in software dynamic translation systems. In: Proc Int Symp on Code Generation and Optimization. San Jose, CA, USA, 2007
14 Bellard F. QEMU, a fast and portable dynamic translator. In: Proc USENIX Annual Technical Conf. Anaheim, CA, USA, 2005
15 Robson D, Strazdins P. Parallelisation of the valgrind dynamic binary instrumentation framework. In: Proc Int Symp on Parallel and Distributed Processing with Applications. Sydney, Australia, 2008
16 Pang Y, Hu W D, Sun L F, et al. Adaptive data-driven parallelization of multiview video coding on multi-core processor. Sci China Ser F-Inf Sci, 2009, 52: 195–205
17 Scott K, Kumar N, Velusamy S, et al. Retargetable and reconfigurable software dynamic translation. In: Proc Int Symp on Code Generation and Optimization. San Francisco, CA, USA, 2003
18 Lu J, Chen H, Yew P, et al. Design and implementation of a lightweight dynamic optimization system. J Instruction-Level Parall, 2004, 6: 1–24
19 Tera-scale Research Prototype: Connecting 80 simple sores on a single est chip ftp://download.intel.com/research/platform/terascale/tera-scaleresearchprototypebackgrounder.pdf
20 Dorsey J, Searles S, Ciraula M, et al. An integrated quad-core Opteron processor. In: Proc of Int Solid State Circuits Conf. San Francisco, California, USA, 2007
21 Wells P, Chakraborty K, Sohi G. Dynamic heterogeneity and the need for multicore virtualization. ACM SIGOPS Oper Syst Rev, 2009, 2: 5–14
22 Stallings W. Operating Systems: Internals and Design Principles 2008. 6th ed. Prentice Hall, 2008
23 SPEC CPU2000 Documentation, http://www.spec.org/osg/cpu2000/docs/