Dynamic Binary Translation and Optimization in a Whole-System Emulator -- SkyEye

Chen Yu[1], Ren Jie[1] ,Zhu Hui[2] , and Shi Yuan Chun[1]
[1]Department of Computer Science and Technology
Tsinghua University, Beijing 100084, P.R.China
[2]Department of Computer Science and Technology
Beijing University of Aeronautics&Astronautics,
Beijing 100084, P.R.China
yuchen@tsinghua.edu.cn

Abstract

This paper presents the design of a high performance whole-system emulator --- SkyEye. Several optimization methods used in SkyEye are proposed and analyzed. By using novel searching strategy for Translated Block (TB), SkyEye save the time to find proper translated block. SkyEye uses Basic Equal Length Unit (B-ELU) method to implement dynamic binary translation. The performance model of B-ELU is built to get the best length of translated block. In order to further reduce the switch time between executing of translated block and searching for translated block, adaptive block linking (ABL) method is designed. Using these methods, SkyEye which simulates ARM CPU based hardware system achieves marvelous performance in experiments.

1. Introduction

Nowadays, binary translation and optimization technologies have achieved a high profile, since there are many famous projects such as the IBM DAISY open-source project, QEMU open-source project, Transmeta Crusoe, and HP Dynamo. Binary translation has several attractive features: firstly, it makes hardware one layer of software, thus guarantees the easy implementation of complex legacy architecture(s) through simple hardware and the introduction of novel concepts without forcing any software changes. Secondly, binary translation enables significant software optimizations that would not be done with hardware alone.

This paper introduces a high performance whole-system hardware emulator –SkyEye [7], on which an unmodified guest operating system (such as Linux, uClinux, eCos, L4) and all related applications can run. SkyEye itself runs on several host operating systems such as Linux, Windows. Currently, SkyEye is capable of simulating most ARM hardware platforms and various peripheral devices, such as NIC, LCD, Flash and UART. Compared with other similar emulators, SkyEye has the following desirable features:

1) Speed optimization. This is beneficial for large-size programs, because it usually takes too much execution time to run these programs on traditional emulators.
2) Flexible combination of hardware simulation. SkyEye can combine different simulated hardware device/CPU components to simulate a whole hardware system.
3) Automatic detection of task switches and function calls. This feature could provide detailed information on energy consumption or performance statistics for each functions in guest operating systems.
4) Hardware statistic function: give statistics on the usage of different SkyEye simulated hardware components (such as Cache) which can provide detailed information for computer architecture researcher.
5) Support for voltage scaling, which enables the energy estimation for those guest operating systems that have dynamical power management [8].

2. Related Researches

SimOS [1] is a MIPS based whole-system emulator. The key part of SimOS is Embra[2] which can run large, real-world programs and commercial operating systems. Embra can handle self-modifying code and can self-host. But the development of SimOS was stopped.

The QEMU machine emulator and dynamic binary translator [4] is an emulation project that was started by Fabrice Bellard. It provides experimental implement-ations for x86, ARM, SPARC and PowerPC source machines running Linux. QEMU runs stably on x86 and PowerPC host machines, and has experimental implementation for Alpha, SPARC, ARM, S390 and IA-64. All of the machine dependent modules in QEMU are written manually. QEMU translates all source machine instructions instead of combining interpretation with the dynamic translation of selected traces.

DELI [3] is a descendant of the original Dynamo dynamic optimization system. DELI is a runtime code translation system that exports an interface for customizing its behavior, which focuses on providing caching and linking services for binary translators and emulators. Its underlying platform is a special VLIW embedded processor, and its primary goal is to support ISA compatibility by flexibly emulating other embedded processors.

The University of Queensland Dynamic Binary Translator UQDBT [5] was an attempt to build a complete dynamic binary translator from specification files. The dynamic binary translator abstracts the source machine instructions into a high-level representation which is then translated into host machine instructions.

The DAISY (Dynamically Architecture Instruction Set from Yorktown) framework [6] is an experimental dynamic binary translator developed at the IBM T. J. Watson Research Center. As the target of their VLIW compiler, DAISY proposes a simple VLIW architecture that is a superset of features of common existing machines and functions. Source machine instructions are translated into VLIW instructions that in turn are interpreted by a VLIW emulator.

3. System Architecture

This section presents the SkyEye architecture and implementation of whole-system simulation. The main goal of SkyEye is to run/debug/analyze guest operating system on it. Figure 1 illustrates the architecture of SkyEye. SkyEye is made of several components:

- CPU emulator (ARM, Blackfin, Coldfire, MIPS)
- Device emulator (LCD, Net IC, Flash, UART, Touch Screen, …)
- Dynamic Binary Translator/Tracer/Fixer (dynamic translating target binary code and optimizing translated host binary code)
- Debugger (debugging the OS on SkyEye in source level with GDB)
- Power Analyzer (analyzing the Power/Energy of Whole System consumption)
- Machine Configuration (according to the Machine Configuration file to assemble the simulated board-level hardware)

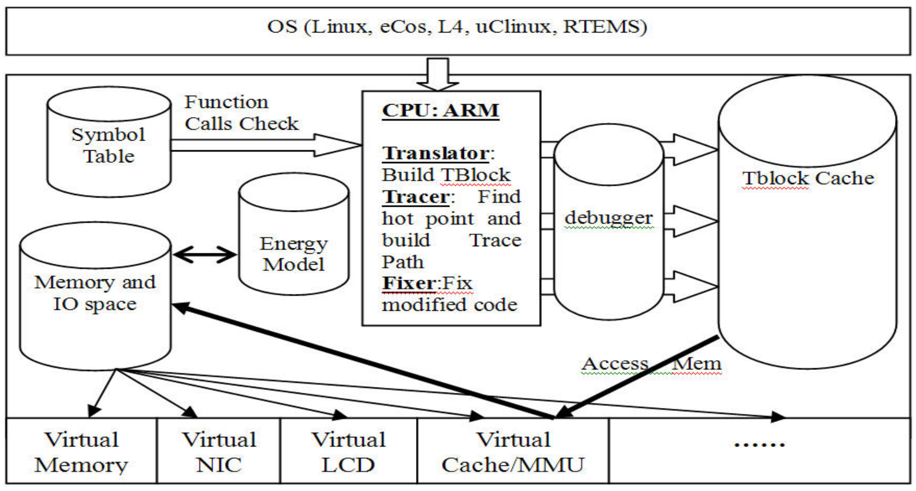

Figure 1 SkyEye Architecture

4.Translation Implementation and Optimization Technologies

4.1. Translation Implementation

SkyEye first parses the Machine Configuration file (skyeye.conf) to get the hardware configuration. Then SkyEye loads executable OS kernel files and file system image that are estimated. After finishing the loading process, SkyEye begins to execute the instructions of OS and applications.

When SkyEye starts to emulate, it first analyzes the operation code and address, then translates each target CPU instruction into a few host instructions which form Translated Block (TB). The detailed process on dynamic binary translation and optimization will be discussed in next section.

If SkyEye needs to estimate the Power/Energy consumption in the whole hardware system, then LDR/STR and other memory access instructions are estimated by "SkyEye Device and Memory Energy Model", while other normal instructions are estimated by "SkyEye Instruction Energy Model". SkyEye Symbol Table Component records the function names and entry addresses. If PC register is equal to the entry address of function, Function Call Check Component is activated and detailed energy consumption information for that function is recorded.

When SkyEye first encounters a piece of target CPU codes, SkyEye Translator Component dynamic translates the codes into translated block (TB). When SkyEye use basic translating method, the length of each target CPU codes is equal. Along with the OS&applications running on SkyEye, the SkyEye Tracer Component monitors the executive frequency of TB, finds the hot spot, and combines several TBs in hot spot executive path into a bigger Super TB adaptively. SkyEye Fixer Component monitors the self-modifying code, and invalidates corresponding TB to guarantee the consistency. If a given TB is invalidated too often because of write accesses, a bitmap representing all the code inside the TB is built. Each time CPU writing into that TB, SkyEye Fixer Component will checks the bitmap to see if the code really needs to be invalidated, so SkyEye Fixer Component avoids invalidating the code parts in TB when only data is modified in TB. In order to improve the performance of SkyEye, we designed several new methods in SkyEye Translator/Tracer/Fixer. The sections below will show these details.

4.2. Performance Model of Translated Unit Constructing

The general methods for translating and constructing unit are Basic Block method and Trace method. Because SkyEye emulates embedded RISC CPUs for which each instruction has the same length, SkyEye use another method — Basic Equal Length Unit method (B-ELU Construction) to build the translated unit. The B-ELU construction method dynamically translates one piece of target code with the same length each time, so it is much simpler than Basic Block or Trace method, and the construction time for the translated unit is less than that of the other two methods. The key factor of B-ELU method is to choose a suitable instruction length for translated unit.

There are four TB accessing modes in the process of SkyEye B-ELU translation. Mode I: Query the recent accessing record of TB, get the corresponding entry address of TB; Mode II: Query the List/Hash Table of translated block, get the corresponding entry address of TB; Mode III: Query the List/Hash Table of translated block, get the corresponding entry address of TB; Mode IV: Translate block at the first time, get the corresponding entry address of TB and update List/Hash Table and recent accessing record. The statistic notation of each mode access time is shown below:

- N_I : The accumulative time of TBA Mode I
- N_{II} : The accumulative time of TBA Mode II
- N_{III} : The accumulative time of TBA Mode III
- $N_{IV.no}$: The accumulative time of TBA Mode IV without the time of self-modified accessing
- $N_{IV.sm}$: The accumulative time of TBA Mode IV with self-modified accessing
- N_{IV} : $N_{IV.no} + N_{IV.sm}$
- N_{all} : The accumulative time of all TBAs

B-ELU method can split the Text (Code) Section Space of user-level process or OS into N (N>=1) equal length TBs. If the length of TB is very short, SkyEye translator has to spend more time on switching the execution of TB and the translating of TB, and N_{all} , N_I , N_{IV} will decrease. But if we set the length of TB to a very large value, some problems will rise, and N_{II} , N_{III} will increase. Three main reasons are shown below:

1. There may exist many accesses whose entry addresses aren't at the beginning or end of the

TB, so SkyEye Translator Component will spend more time on partial translation of TB.

2. There may be several non-instructions in large TB, so SkyEye Translator Component has to do several unnecessary translations.
3. Large TB could include Text Section, Data Section, BSS Section, so SkyEye Fixer Component for self-modified code might make wrong judgment, and do some unnecessary TB re-translation.

In order to find the best Search strategy for TB, we design and compare three different search strategies for TB. The implementation of each strategy is shown in Figure 2, Figure 3 and Figure 4. The primary concern is how to reduce searching time of access address for TB. The first strategy is to record the translated TB in a list, and search the list every time. The second strategy is to use hash table to replace list and use recent access address record to store the last access address of TB. The third strategy is to record all possible access address in a translated TB and the last access address of TB.

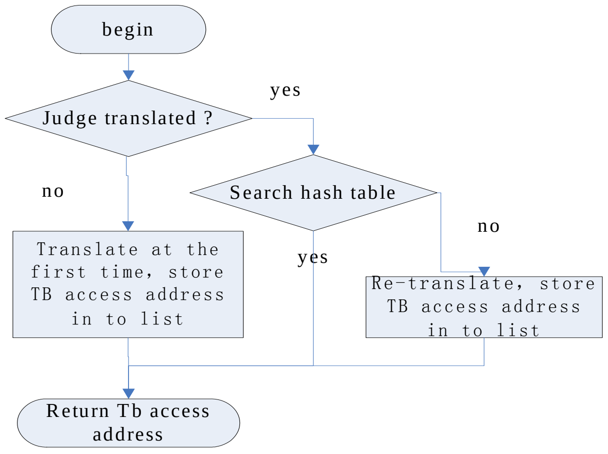
begin
Judge translated ?
yes
no
Search hash table
no
yes
Translate at the first time, store TB access address in to list
Re-translate, store TB access address in to list
Return Tb access address

Figure 2 search strategy 1 for TB

The speed of each strategy is: Strategy 1 < Strategy 2 < Strategy 3. The memory space consumed by each strategy is shown below:

- Strategy 1: 12* N_I +16*(N_{III} + N_{IV})
- Strategy 2: (8+8+TB_LEN)* N_{IV}
- Strategy 3: (8+8+TB_LEN)* N_{IV}

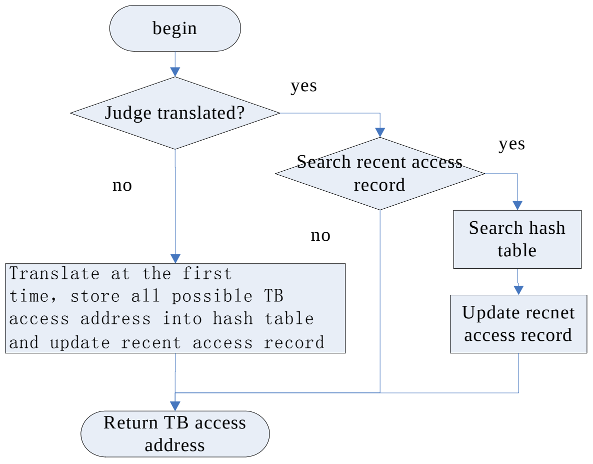
begin
Judge translated?
yes
no
Search recent access record
yes
no
Search hash table
Update recnet access record
Translate at the first time, store all possible TB access address into hash table and update recent access record
Return TB access address

Figure 3 search strategy 2 for TB

In order to find the suitable length of translated block (TB), the -ELU performance model was introduced. Some notations are presented first. The first parts of notations are the number of different process for TB; The second parts of notations are the time of different process for TB.

M: The total number of TBA Modes, which represents the switch of TB translation. TBA i means type i of TB (i=1,2,3,4); M_1: number of recent queries for TB access address record; M_2: the number of query for chain list or hash table; M_3: the number of partially translation for TB; M_4: the number of complete translation for TB which isn't because of self-modified code; M_{sm}: the number of complete translation for TB which is because of self-modified code.

T_{DBT}: The accumulative time of dynamic translating B-ELU; $T_{T\&Q}$: The accumulative time of translating & querying in four TBA Modes; T_i: The accumulative time of TBA mode i, （i=1,2,3,4）; $\overline{T_1}$: The average time of search recent access record; $\overline{T_2}$:The average time of search list or hash table; $\overline{T_3}$: The average time of partially translating TU; $\overline{T_4}$: The average time of complete translating TU; $\overline{T_{CPU}}$: The average time of translating one target CPU instruction; L_{TU} : The length(in byte) of TB; T_E: The accumulative executive time of TB; T_S: The accumulative switch time of TB; $\overline{T_S}$:The average switch time of TB.

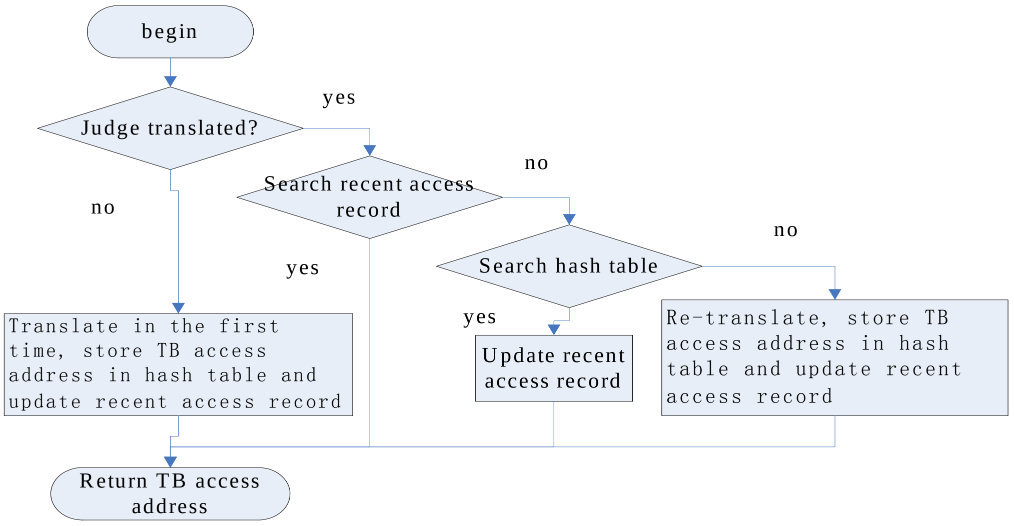
begin
Judge translated?
yes
no
Search recent access record
no
yes
Search hash table
no
yes
Translate in the first time, store TB access address in hash table and update recent access record
Update recent access record
Re-translate, store TB access address in hash table and update recent access record
Return TB access address

Figure 4　search strategy 3 for TB

The main executive cycle of SkyEye includes: Searching TB----Translating TB----Executing TB. So we can get the total time of SkyEye's dynamic binary translation using the equation shown below:

$$T_{DBT} = T_{T\&Q} + T_S + T_E \qquad (1)$$

$$T_{T\&Q} = \overline{T}_1 \times (N_I + N_{II} + N_{III}) + \overline{T}_2 \times (N_{II} + N_{III}) + \overline{T}_3 \times N_{III} + \overline{T}_4 \times (N_{IV.no} + N_{IV.sm}) \qquad (2)$$

$$T_S = \overline{T}_S \times (M + M_{SM}) = \overline{T}_S \times (\sum_{i=1}^{4} M_i + M_{SM}) \qquad (3)$$

According to equation(1)--(3), wc can get the B-ELU performance equation:

$$T_{DBT} = \overline{T}_1 \times (N_I + N_{II} + N_{III}) + \overline{T}_2 \times (N_{II} + N_{III}) + \overline{T}_3 \times N_{III} + \overline{T}_4 \times (N_{IV.no} + N_{IV.sm}) + \overline{T}_S \times (\sum_{i=1}^{4} M_i + M_{SM}) + T_E \qquad (4)$$

The performance equation for the strategy 1 is:

$$T_{DBT} = \overline{T}_2 \times (N_I + N_{II} + N_{III}) + \overline{T}_3 \times N_{III} + \overline{T}_4 \times N_{IV.no} + \overline{T}_4 \times N_{IV.sm} + \overline{T}_S \times (\sum_{i=1}^{4} M_i + M_{SM}) + T_E \qquad (5)$$

The performance equation for the strategy 2 is:

$$T_{DBT} = \overline{T}_1 \times (N_I + N_{II} + N_{III}) + \overline{T}_2 \times (N_{II} + N_{III}) + \overline{T}_3 \times N_{III} + \overline{T}_4 \times N_{IV.no} + \overline{T}_4 \times N_{IV.sm} + \overline{T}_S \times (\sum_{i=1}^{4} M_i + M_{SM}) + T_E \qquad (6)$$

The performance equation for the strategy 3 is:

$$T_{DBT} = \overline{T}_1 \times (N_I + N_{II} + N_{III}) + \overline{T}_2 \times (N_{II} + N_{III}) + \overline{T}_4 \times N_{IV.no} + \overline{T}_S \times (\sum_{i=1}^{4} M_i + M_{SM}) + T_E \qquad (7)$$

According to the equation (5), equation (6) and the fact $\overline{T}_2 \times N_I >> \overline{T}_1 \times (N_I + N_{II} + N_{III})$, Strategy 2 searches the recent accessing address record in very short time, therefore it doesn't need to search the relatively slow hash table every time and could obtain better performance than strategy 1. According to the equation (6) and (7), Strategy 3 saves the time $\overline{T}_3 \times N_{III}$, so it get has a better performance than strategy 2.

In order to get the parameters in these equations, we use GNU gprof to test uClinux+MiniGUI running on SkyEye and estimate the times of the four TUA modes. The result is shown in Table 1.

From the data in Table 1, we can calculate the value of $(N_I + N_{II} + N_{III}) / N_{IV.sm}$, which represents the average utilization of TB. If the utilization of TB is the only concern, then the longer TB is a better choice. However it also shows the value of $N_{IV.sm}$ changes dramatically when the length of TB is longer than 16KB. The reasons were already presented in Section 4.2.

We did several measurements by running uClinux and MiniGUI applications on SkyEye, and get those parameters in equations.

It shows that the search strategy 3 is SkyEye's best B-ELU search strategy. By assigning the parameters with the above data, $\overline{T}_4$ becomes

$$\overline{T}_4 = \overline{T_{CPU}} \times (L_{TU} / 4) \qquad (8)$$

Table 1 the experiment values of NI/ NII / NIII / NIV.no / NIV.sm / Nall

Length of TB	N_I	N_{II}	N_{III}	$N_{IV.no}$	$N_{IV.sm}$	N_{all}
4	26096887	0	0	53039	0	26149926
32	5451358	1589436	5006	8360	0	7054160
128	2726851	2752042	6372	2735	0	5488000
1024	1715849	2881449	7262	596	0	4605156
4096	1496608	3057402	7510	223	0	4561743
32768	1484681	3063538	7646	42	1	4555908
65536	1482669	3065527	7676	24	8	4555904

Because $\overline{T_1}$ and $\overline{T_2}$ is respectively two magnitude smaller than $\overline{T_{CPU}}$ and $\overline{T_S}$, $\overline{T_1}$ and $\overline{T_2}$ could be ignored. Therefore the equation 7 for strategy 3 can be simplified as:

$$T_{DBT} - T_E = \overline{T_{CPU}} \times (L_{TU} / 4) \times (N_{IV.no} + N_{IV.sm}) + \overline{T_S} \times (\sum_{i=1}^{4} M_i + M_{SM}) \qquad (9)$$

Because the execution time of target CPU instructions is irrelevant to the length of TB, T_E could be assumed as a constant. When the measured parameters are assigned into equation (9), $T_{DBT} - T_E$ will be minimal with the length of TB in [256, 4096]. Therefore the best region of the length of TB is [256, 4096].

Table 2 the measured parameters in equations (1)~(7)

T1	N1	T2	N2	T4	Length of TB/4	N4	Ts	N+NWB
1.06	26096887	0	0	0.023	1	53039	26.53	26149926
0.71	14676940	0.04	496755	0.19	2	27856	14.33	14704796
0.57	9564991	0.11	1207263	0.15	4	15073	9.63	9580064
0.48	5772217	0.16	2180475	0.14	16	4735	5.77	5776952
0.32	4902964	0.38	2732207	0.2	64	1613	5.05	4904577
0.27	4615278	0.36	2882779	0.26	128	971	4.86	4616249
0.27	4569921	0.25	3021458	0.25	512	359	4.92	4570280
0.20	4561520	0.17	3064912	0.33	1024	223	4.50	4561743
0.17	4561025	0.26	3065104	0.36	2048	128	4.94	4561153
0.18	4560619	0.17	3065565	0.53	4096	73	4.91	4560692
0.21	4555865	0.23	3071184	0.42	8192	43	4.56	4555908
0.17	4555872	0.28	3073203	0.75	16384	32	4.64	4555904
$\overline{T_1}$ =5.25744x10^-8		$\overline{T_2}$ =8.97373x10^-8		$\overline{T_{CPU}}$ =1.64x10^-6			$\overline{T_S}$ =1.02625x10^-6	

From the SkyEye B-ELU performance equation, we can analyze the factors that affect the translation performance. When we use B-ELU with best strategy and Best Length of TB, the result is good. But we also have detected that the time of switch time of TB executing turns into the performance bottleneck of SkyEye. There for, Adaptive Block Linking method is used to solve the problem.

4.3. Adaptive Block Linking

When SkyEye sets 4096 Byte as the length of TB and uses searching strategy 3 for TB, we have detected that the switch time of TB executing became the performance bottleneck. In order to reduce the switch time, adaptive block linking (ABL) method is designed and realized in SkyEye Tracer Component. ABL method can reduce the switch time of TB and speed up the execution. The process of ABL method is:

1. After each TB is executed, SkyEye Tracer Component uses the simulated target CPU Program Counter register and other information of the simulated target CPU state to find the next executing TB in hash table.
2. If the next executing TB has not been already translated, SkyEye translator will dynamically translate the next TB. Otherwise, if the current TB and next TB doesn't have self-modified indication, then a direct jump at the end entry of the current TB is made, and the hash table is updated.
3. If the current TB and next TB were linked by the direct jump, they form a new bigger TB --- Super TB.
4. When there are self-modifying actions in Super TB, the first basic TB be modified will be found in the Super TB and the Super TB will be divided into several basic TBs. Finally, SkyEye Translator Component re-translates the modified basic TB and updates hash table.

5. Experiments and Results

The Experiment environment includes a PC with 2GHZ P4 CPU with 256MB memory. We run uClinux-2.6.x+applications on SkyEye with different strategies and different length of TB. We also have modified QEMU-0.8.0 to run ARM Linux-2.6.x+applications and test the performance of QEMU. Figure 5 compares the performance of SkyEye using different methods and QEMU. SkyEye and QEMU were compiled by GCC-3.3 with CFLAGS='-pg …', then the performance statistic data can be gathered by gprof.

From the data collected by gprof, SkyEye using B-ELU with 4 bytes TU is about 1 times faster than SkyEye using interpreting methods. If SkyEye chooses 4096 byes TU and searching strategy 3, it is about 6 times faster than SkyEye using interpreting methods. If SkyEye adds ABL method to reduce the switch time, it is about 10 times faster than SkyEye using interpreting methods, and is also faster than QEMU. We also noticed that SkyEye used almost all memory (196MB) in PC for TB cache, but QEMU occupied about 64 MB memories.

The experiment results show that SkyEye with B-ELU (4096 bytes) + searching strategy 3 + ABL is fastest. QEMU uses optimized methods including basic block as TB, direct block chaining, condition code optimization and register allocation. The performance of QEMU was also very good.

6. Conclusions

In this paper, we present an implementation of an emulator—SkyEye. SkyEye uses novel searching strategy for translated block (TB) to reduce the search time for TB. The performance equations of dynamic binary translation were built and the best TB length was calculated using measured parameters. The adaptive block linking (ABL) method was designed to delimit the unnecessary switch time of TB executing. With the above strategy and methods, SkyEye got a good performance in experiment The future work will focus on designing the distributed SkyEye Tracer to improve the performance of SkyEye further., designing loop structure recognizer to optimize the loop structure in target OS&Application. The debugger support on SkyEye will be added to help developers to develop or debug system software more easily.

Acknowledgments. This paper was supported by the National High-Tech Research and Development Plan of China (No.2003AA1Z2090), National Natural Science Foundation of China (No. 60203024), and Tsinghua University Research Found (No.JC2003021), University. Doctorial Research Foundation, No.20050003048.

The SkyEye distribution is available online under GPL license. Please refer to the SkyEye home site for more information about this project:
http://www.skyeye.org

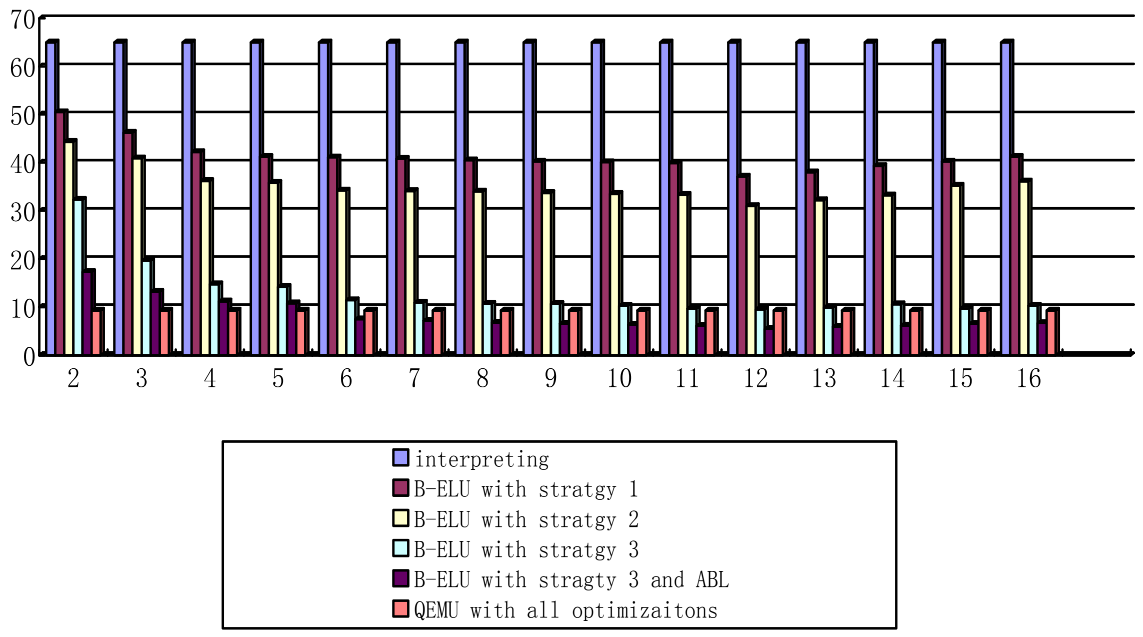

Figure 5 the performance of translating&optimizing methods in SkyEye and QEMU the value of X axis: the length of TB= 2^(x) using B-ELU in SkyEye the value of Y axis: executing time (second)

7. References

[1] Rosenblum, M., Herrod, S., Witchel, E., AND Gupta, A. 1995. The SimOS approach. IEEE Parallel and Distributed Technology, 4(3), 34-43.

[2] Witchel, E., AND Rosenblum, M. 1996. Embra: Fast and flexible machine simulation. In Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 68-79.

[3] Desoli, G., Mateev, N., Duesterwald, E., Faraboschi, P., AND Fisher, J. A. 2002. DELI: A new run-time control point. In Proceedings of the 35th International Symposium on Micro architecture (MICRO '02), 257–268.

[4] David Ung and Cristina Cifuentes. Optimizing Hot Paths in a Dynamic Binary Translation. Workshop on Binary Translation, 2000. 38- 139

[5] Fabrice Bellard. The QEMU CPU Emulator, 2004. http://fabrice.bellard.free.fr/qemu/.

[6] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In ISCA, pages 26–37, 1997. 41

[7] Chen Yu, etc., SkyEye Emulator, http://www.skyeye.org

[8] Kang Shuo, Wang Hua yong, Chen Yu, “An Energy-awared Simulator for Energy Co-estimation in the Embedded System”, The 2004 International Conference on Embedded Software and System, LNCS.,2004.10