Scientific Research and Essays Vol. 7(8), pp. 939-946, 29 February, 2012
Available online at http://www.academicjournals.org/SRE
DOI: 10.5897/SRE11.1790
ISSN 1992-2248 ©2012 Academic Journals

Full Length Research Paper

Planning for fast DBT in distributed virtual execution environment

Yindong Yang and Erzhou Zhu*

Department of Computer Science and Engineering, Shanghai Key Laboratory of Scalable Computing and Systems Shanghai Jiao Tong University, Shanghai, P. R. China.

Accepted 22 November, 2011

Virtualization via dynamic binary translation is essentially an emulator. The main advantage is that the CPU of the guest does not have to be the same as the CPU of the host. Although, dynamic binary translators (DBT) are gaining popularity and offering a promising future in the modern virtual executive environments, the requirements of DBTs' processing and memory resources have seriously hampered the performance of host platforms. For cloud computing seems to offer incredible lightning-quick processing power and unlimited storage. In this paper, we propose a novel distributed DBT system-DistriBit for resource-limited thin clients computing. Meanwhile, we study the effects of the number of virtual registers and trace length to improve the performance of DistriBit. Our results demonstrate that improving these two factors (the number of virtual registers and trace length) may help to improve program speedup by up to 1.4 to 3.9*x* and 2 to 3*x* for certain benchmark programs.

Key words: Virtualization, dynamic binary translation, DBT, cloud computing, distribit.

INTRODUCTION

Cloud computing (Michael et al., 2009) is a fashionable technology which uses the internet and powerful remote servers to acquire data and applications. Furthermore, cloud computing allows clients to use applications without installation and access their personal files or requisite resources at any computer with internet access. This technology allows for much more efficient computing by centralizing storage, memory, processing and bandwidth. Virtualization is a core technology for enabling cloud resource sharing (Andrés, 2009), and it uses a single data processing system to run multiple operating systems (for example, Linux, Windows) or applications based on different architectures (for example, *x*86, MIPS, POWER).

Both in virtual execution environments and cloud computing environments, DBTs are usually considered as a kind of processed virtual machines, which support user applications with a virtual application binary interface environment (Jim and Ravi, 2005). There is no need to recompile the source code, since DBTs facilitates the support of running heterogeneous code on a wide range of architectures. A DBT is originally developed to support program binaries compiled to a different instruction set rather than the one executed by the hosts hardware dynamically, and more and more DBTs have been used as optimizers or dynamic binary analysis tools nowadays (Vasanth et al., 2000; Chi et al., 2005; Sorav and Alex, 2008).

Although, DBTs are very attractive and ubiquitous, they also exhibit major shortcomings. First, the DBT itself is very complex; developing a complete process-level DBT from the scratch always takes a lot of manpower and material resource, not to mention the development of system-level DBT (Cristina and Mike, 2000). Secondly, state-of-the-art monolithic DBTs rely on the monolithic architecture of their ancestors and highly machine dependence (John and Gerald, 1994), so they are difficult to port across the diverse architectures and platforms found in a typical network. Thirdly, DBTs' processing and memory requirements often make host machine unbearable (Borin and Youfeng, 2008),

*Corresponding author. E-mail: yasakaezzhu@sjtu.edu.cn.Tel: +86-15900619365, +86-21-23119272. Fax: +86-21-23112489.

especially for those resource-limited thin clients. As a result, directly using monolithic DBTs on resource-limited thin clients may be unsuitable and impractical. On the other hand, cloud computing offers an immediate access to large numbers of the world's most sophisticated supercomputers and their corresponding processing power. The power of cloud computing could compensate for some shortcomings of thin clients, and the thin clients may benefit from it in cloud computing environments.

We have developed a distributed DBT–DistriBit for cloud computing environments. In a DistriBit system, some works which need substantial processing and memory requirements, such as code translation and optimization are factored out of thin clients and located on powerful servers in cloud. By adopting this approach, thin client can reduce the hardware configuration and concentrate on execution. Surprisingly, we found that DistriBit rarely offers performance advantages over existing monolithic DBT. In some cases, we ascribe this situation to high network transmission overhead and cost of virtual intermediate instructions' translation. Overall, in this paper, we focus on the latter and study how optimization factors are related to virtual intermediate instructtions and as such affects the running performance. Furthermore, we hope our results encourage software designers to support distribute DBT techniques rather than seek to replace them. We believe the benefits of distribute DBTs to cloud computing are compelling.

In summary, our aim in this work is to demonstrate the ability to improve the running performance of DBT in distributed virtual execution environment. Towards this end, we make the following main contributions in this paper: (1) an introduction of the distribute DBT system–DistriBit, focusing on its virtual intermediate instructions (V-IIS)–a powerful but previously little-studied component; (2) a quantitative analysis of optimization factors the number of virtual registers and trace length.

RELATED WORK

Traditional monolithic DBT

The UQBT (Cristina, 2000), is a reusable, component-based binary-translation framework which allows engineers quickly and inexpensively migrate existing software from one processor to the other. Different from V-IIS, UQBT uses two intermediate instructions to transform the source code into a high-level representation. The more middle transformation, the more performance will be lost. That is the reason why papers about UQBT focus on concept and not performance.

Dynamo (Vasanth, 2000), a dynamic optimization system for PA-RISC, developed by Hewlett-Packard Labs showed good performance. Unfortunately, many similar systems with intermediate languages have been built but little success has been reproduced ever since Dynamo.

Distributed virtual computing

DVM (Emin et al., 1999) is a distributed virtual machine designed for heterogeneous clusters of networked computers. DVM uses Java virtual machine to run on *x*86 or DEC Alpha server to supply services to thin clients. DVM could reduce resources requirement on thin clients, improve site security through physical isolation and increase the manageability of a large and heterogeneous network without sacrificing performance, but the limitation is that DVM only supports existing Java-enable clients not other Java-unable clients. In contrast, DistriBit is language independent and can support heterogeneous software without recompiling the source code.

Emin (1998) designed and implemented a system similar to DVM, services that perform rule checking and code transformation are factored out of thin clients and located on severs. However, these severs are located in enterprise-wide network, can't spreading widely.

General trace optimization

Many works which use traces for exploring program optimization opportunities have been done. To our knowledge, the work closest to ours has been done in (Brian, 2004) and (Andres, 2009). Fahs et al (Brian, 2004) demonstrated the potential of dynamically-applied classical optimizations as a function of trace length. However, their results are based on ideal circumstances, and not easily compared to existing dynamic optimization systems. Moreover, they did not study the potential of virtual registers.

Andres (2009) described their type specialization algorithm and trace compiler, which translated a trace from an intermediate representation to optimized native code in two linear passes. Their experiments showed that on programs amenable to traces, they achieved speedups of 2*x* to 20*x*. Different from ours, their programs are currently executed via interpretation not dynamic binary translation.

METHODOLOGY

We now move on to begin with a necessarily brief overview of the distributed DBT system–DistriBit, followed by a detailed description of its virtual intermediate instruction set–V-IIS.

Architecture overview

Considering that code translation and optimization always consume a lot of computation and memory resources, we address the problems of monolithic DBTs by proposing a novel distributed DBT–DistriBit (Haibing et al., 2010) based on function factoring. The DistriBit supplies a virtual executive environment wherein a powerful server could provide code translation, code optimization services and an unbounded code cache for thin client. With the help of this powerful server, thin client does not need to perform code translation but still

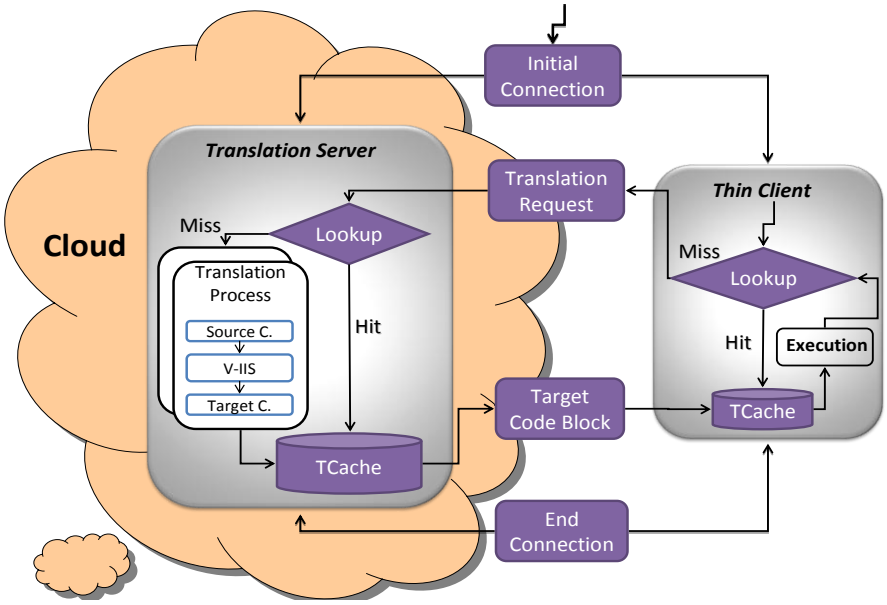


Figure 1. The workflow of a DistriBit system.

Table 1. The entire V-IIS instructions.

Type	Instruction name
register state mapping	GET PUT
Memory access	LD ST
Data transfer	MOV LI
Arithmetic/logic	ADD SUB AND NOT XOR OR MUL MULU DIV DIVU SLL SRL SRA CMP SEXT ZEXT
Control transfer	JMP BRANCH
Special	HALT SYSCALL CALL

can get the optimized code from the server whenever necessary. In this manner, the burden of hardware requirements for thin client will be reduced. Figure 1 illustrates distributed DBT architecture–DistriBit. For instance, a DistriBit system consists of a translation server and a thin client that communicates over a network using TCP/IP transportation protocol. This translation server could be one of those powerful servers located in a cloud, and the thin client could be any thin client device, like mobile phone, sensor, etc. In general, the connection between them can be wired or wireless.

DistriBit is a general-purpose distributed DBT that could supply heterogeneous code translation and optimization services to thin clients. DistriBit uses a translator (Yindong et al., 2010) located on a powerful server to response to thin clients' translation requests. The *t*ranslator that DistriBit adopts differs from other commercial machine dependent on DBTs which only can translate one kind of source code to another specific target code. This translator is a multi-source and multi-target DBT which can translate different kinds of source code to different kinds of target code, and undoubtedly reduce the number of DBT systems on server. By now, this versatile translator has run on IA-32 and POWER, and has supplied MIPS-IA32, SPARC-IA32 and SPARC-POWER code translation to users. Moreover, this translator is machine independent and can easily add other source ends and target ends according to the actual needs, so it may provide better and comprehensive services for thin clients.

V-IIS instructions

In order to reduce the complexity of translation, DistriBit's translator does not translate source code to target code directly. We design a set of virtual intermediate instructions called V-IIS and let translator firstly translate source code to intermediate instructions, then to target code. Therefore, the V-IIS makes this translator easily retargetable and extensible. Similar to V-IIS, LLVA (Vikram et al., 2003) is a low-level virtual instruction set designed for life-long (including compile, link and run) code optimization. Though, as a low-level virtual instruction set, LLVA contains enough rich high-level information (for example, control flow, data flow and data dependence) required by optimization algorithms, and these information usually can be extremely difficult to extract from native machine code. However, Different from our research, LLVA's design goal and object code cannot be applied to the analysis of low level executable binary code. V-IIS comprises six kinds of basic instructions which are compatible with most popular ISAs. They include arithmetic/logical, control transfer, data transfer, and memory access; register state mapping and special instructions. Table 1 shows all virtual intermediate instructions of V-IIS.

Research concept

There are a number of factors that impact a DBT's performance improvement, including the virtual registers' allocation and the instruction traces selected. Virtual registers hold data in binary code, which makes virtual, register allocation an important factor in dynamic

```
1  Virtual_Register_Allocation (trace t, int max)
2  {
3    VR_record = Init (max); /* VR information*/
4    ins = Get_First_Instruction (t);
5    while (ins)
6    {
7      for each r in ins
8      {
9       if ( ture == Not_Allocated (VR_record, r)
10        {
11          r = Assign_Virtual_Register (VR_record, r);
12        }
13      else
14        {
15          r = New_Virtual_Register (VR_record, r);
16        }
17     }
18    ins = ins->next;
19   }
20 }
```

Figure 2. Virtual register allocation on a trace.

binary optimization. Traces define the optimization unit and the scope of the code. Hence, trace selection in dynamic binary optimization is as critical as region selection in static optimization. Therefore, all of these optimization factors have to be handled carefully for a DBT to gain performance improvement.

Number of virtual registers

Surprisingly, according to the initial experimental results, we found that DistriBit rarely offers performance advantages over existing monolithic DBT. One main reason for this situation is that we introduce a virtual intermediate instruction set called V-IIS during the translation process.

The causalities incurred when using V-IIS are due to the following reasons. First, independent V-IIS could be collected and then reused for new objects, dramatically reducing the cost of handling machine idiosyncrasies, and a developer is able to concentrate on writing descriptions of properties of target machines instead of having to rewrite the DBT itself. Second, V-IIS allows developers to performing machine-independent analysis and optimization on the fly, such as inserting code to do basic block counting and profiling. We can also understand the behavior of running programs by recording dynamic memory accesses, branches taken or not, and instruction traces from V-IIS.

Despite its advantage, V-IIS exhibits major shortcoming and that is loss in efficiency. As a process virtual machine, DBTs similarly add another layer on host machine and inevitably lose some performance during translation. The more the layer, the more performance will be lost. The use of V-IIS exacerbates this situation and leads to more performance loss. In general, virtual registers have a greater impact on performance, because they hold the data that translator manipulates. Register allocation determines the length of live intervals, which further affects data availability. Hence, register allocation can impact many optimization measures. Register allocation includes allocation of virtual registers and physical registers. In this paper, we focus on virtual register allocation. Normally, the benefit of dynamic optimization can be limited by a DBT's virtual register allocation algorithm. For example, common sub expression elimination can be applied only when the result of a sub expression is held by a virtual register and its live interval reaches another instance of the sub expression. During the design process of V-IIS, like most virtual instruction sets (for example, LLVA (Vikram et al., 2003)), we set V-IIS with unlimited numbers of 32-bit virtual integer registers. However, the concept of "the unlimited numbers" is vague, also imprecise. How does the number of virtual registers affect the performance? This critical question has not been addressed before and we aim to address this lack of knowledge by using approximation in approaching the result of ideal virtual register allocation.

Our method of virtual register allocation on a trace is to aggressively allocate new virtual registers, and the number of virtual register is set as a configurable parameter and increased from the minimum (that is, larger than or equal to the number of host's physical register) to the maximum step by step. Instead of re-using virtual registers for other values, the goal of aggressively allocating is to extend the availability of the results held by virtual registers. Figure 2 shows the virtual register allocation algorithm. First, line 2 initializes the data structure with virtual register information (*VR_record*) and the number of virtual registers (maximum). Secondly, every virtual register *r* used by instruction *ins* on the trace *t* must go through several steps. If *r* has not been allocated before, this virtual register number is assigned as in line 11 and the allocation information of *r* is recorded in *VR_record*; otherwise, a new virtual register will be used as in line 15. When maximum is smaller than or equal to the number of physical registers, things become simple. Finally, if the number of virtual registers exceeds the number of physical registers, we use "next-use" register allocation algorithm to handle this problem. Our virtual register allocation scheme is effective and extending the availability of useful results, which exposes optimization opportunities and helps an optimizer detect and remove redundant computations.

Trace length

Indeed, basic block linked as a trace is an effective optimization measure, which links direct and indirect branches between blocks. Optimization opportunities emerge when a trace is formed into a single entry and multiple exits superblock, where a variable in the original program may become constant and a partial redundancy may change to a full redundancy. It has been used in many DBT systems (for example, Dynamo (Vasanth, 2000)) and it helps speed up the performance even up to 10 times. Theoretically, performance speedup only happens when trace length reaches a certain number of basic blocks. For example, assume that *freq* (*t*) denotes the total execution frequency for path *t*. So, for a set *T* of paths we define the flow of *T* as:

$$freq(T) = \{ freq(t) \mid t \in T\} \quad (1)$$

A path *t* is regarded as a trace if and only if *freq (t)* is greater than some hot threshold θ (that is, a piece of translated code in the code cache is executed "too often"). The set of traces with respect to θ is defined as:

$$Trace_\theta = \{t \mid freq(t) > \theta\} \quad (2)$$

Furthermore, assume a path *t* is predicted after it has executed η times. η is called the prediction delay and the execution flow captured by this prediction is: $freq(p)$-η. The hit rate for *T* now results as:

$$HitRate\ (T) = [\text{freq}\ (T \cap Trace_\theta) - |\ (T \cap Trace_\theta)| \times \eta] \times 100 \,/\, freq\ (Trace_\theta) \quad (3)$$

The improvement of the hit rate needs minimizing the η of (3), and then the trace execution overhead (verify the targets of returns and indirect branches) is significantly reduced. As a result, a long trace is likely to contain more opportunities for optimization than a short trace. However, traces should also follow execution flow, and the benefit of the optimization opportunities is only gained when the traces execute.

```
1  Trace_Create (trace t, int threshold)
2  {
3    while (t.blockCount < threshold)
4     {
5       newBlockAdded = False ;
6       for each bblock after trace t
7        {
8          newBolckAdded = True ;
9          Trace_Linking (bblock) ;
10           bblock = bblock -> next ;
11           blockCount ++ ;
12       }
13       if ( newBlockAdded == False)
14       {
15          break ;
16        }
17     }
18  }
```

Figure 3. Block linking as a trace.

Table 2. Experimental setup.

	IA-32	POWER
Processor	Intel® Xeon® 2.00GHz(quad-core)	POWER® 6 1-core 4.2GHz
Memory	4GB	1GB
OS	CentOS 5.1	Linux 2.6
Compiler	gcc-4.2.3	
Binary	static	

Therefore, traces need to be carefully selected both to expose opportunities and to realize optimization benefit. We try to figure out that when the speedup saturates once trace length reaches a certain number of basic blocks.

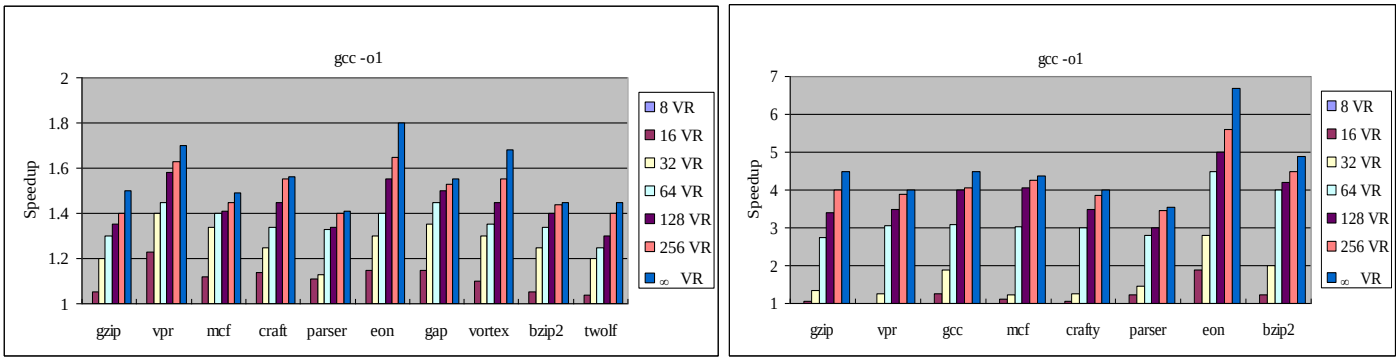
We consider traces with different lengths to find the point at which their performance saturates. The lengths of traces are gradually increased until the set point is reached. Figure 3 abstracts the block linking algorithm. Given a trace, we set threshold as a limit of its number of basic blocks. First, line 5 initializes the trace *t* holding nothing. Second, lines 6 to 12 keep increasing the length of trace *t* along executing paths until the threshold is met. Otherwise, lines 13 to 16 exit trace construction.

RESULTS AND DISCUSSION

Here, we present the results of the optimization measures through experimentation with DistriBit. Through this evaluation, we want to assess the following two issues. First, how does the number of virtual registers affect the overall DistriBit's running performance? We compare its performance with a different number of virtual registers. Moreover, we need to determine the number of virtual registers that provides a reasonable approximation to deal with different optimization levels (for example, -o3, -o1). Secondly, once the optimal number of virtual registers is determined, the question is whether it will affect other optimization measures, such as block linking (that is, trace length). Taking into account the translation of the procedures between same platforms is meaningless, we choose two typical pairs (MIPS–IA-32 and IA-32–POWER), and make an evaluation of them. MIPS and POWER are reduced instruction set computer ISAs with a lot of registers. The opposite situation is that IA-32 belongs to complex instruction set computer ISAs with a small number of registers. The huge differences between them could illustrate the problem better. We adopt the popular method of testing, which is running the SPECint2000 with test inputs. Other missing programs are due to failed translations. The experimental setup is described in Table 2.

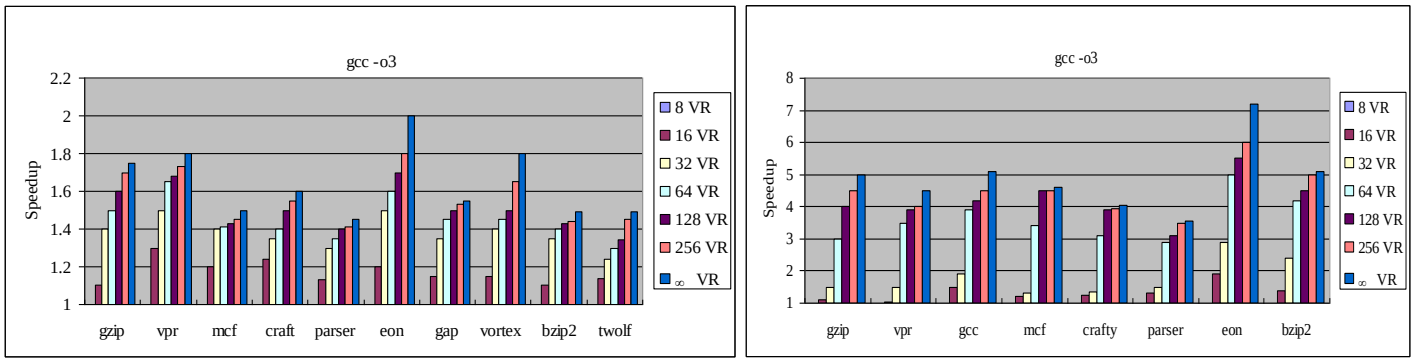
Virtual registers number

It is generally recognized that the number of physical registers has a tremendous effect on performance. In fact, virtual register also has a potential impact on the



(a) MIPS—IA-32 (b) IA-32—POWER

Figure 4. Speedup with a different number of virtual registers. The bars represent performance relative to 8 VR situation (initialized to 1), higher is better. (Benchmarks are compiled by gcc -o1).



(a) MIPS—IA-32 (b) IA-32—POWER

Figure 5. Speedup with a different number of virtual registers. The bars represent performance relative to 8 VR situations (initialized to 1), higher is better. (Benchmarks are compiled by gcc -o3).

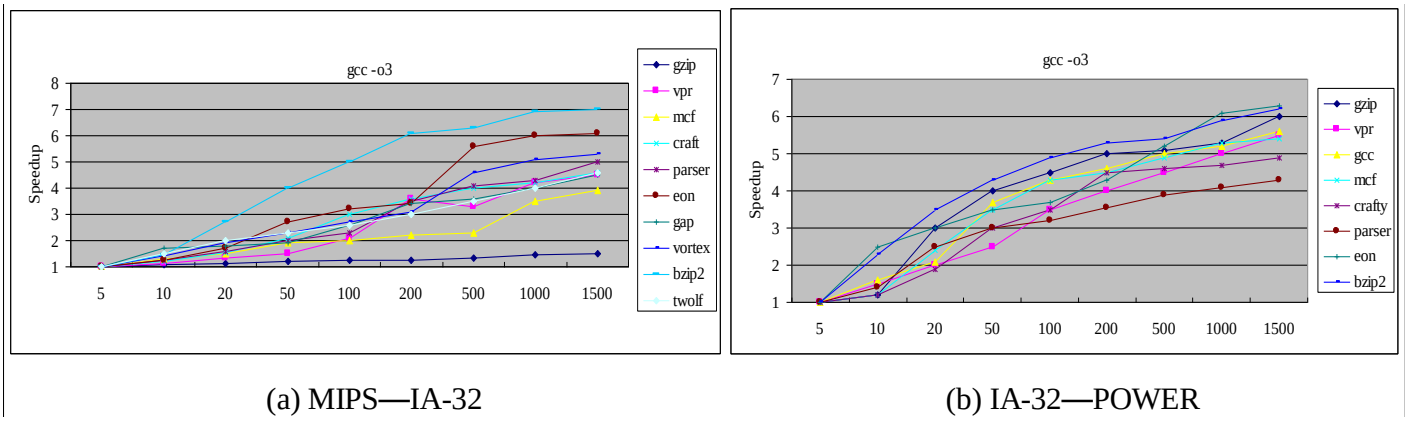
performance, especially in dynamic binary translation process. We collect program speedups from DistriBit when the virtual registers are equipped with 8 to ∞ in a distribute environment. When the number of virtual registers is ∞, it means that we do not specify a particular number, and this number depends on the actual situation of running.

We present the results for two compiler optimization levels (that is, -o1 and -o3) because the binary executable code is sensitive to static optimization level, and then influences the results. Due to space limitations, we think these two moderate optimization and aggressive optimization levels are sufficient in describing the problem. Therefore, we aim to avoid overestimates. From Figures 4 and 5, as expected, the speedup grows with the number of virtual registers increasing. This phenomenon shows that DistriBit's performance can indeed benefit from more virtual registers.

In fact, there is no upper bound in the number of virtual registers, and it is difficult to determine how many virtual registers are close to ideal register allocation. When studied carefully, a close look at Figures 4 and 5 show that every increase in the number of virtual registers helps performance improvement and going from 16 to 32 and 32 to 64 is the largest boost respectively. Hence, we conclude that 32 or 64 virtual registers represent considerable benefit of increasing the number of registers, though 32 and 64 registers are not overwhelmingly many. The average potential speedup is 1.4 and 3.9*x*, and it is not surprising that the potential of performance improvement is sensitive to static optimization.

Trace length

Dynamic binary translation is based on traces which are a sequence of basic blocks, typically chained along



(a) MIPS—IA-32 (b) IA-32—POWER

Figure 6. Speedup on traces with a different number of basic blocks. The lines represent performance relative to 5 situations (initialized to 1), higher is better. (Benchmarks are compiled by gcc -o3).

frequently executing code paths. Traces are formed between the processes of translating virtual intermediate code into target code. The quality of trace forming is affected by the speed of virtual intermediate code generating. So, we need to find a trace length that is long enough to approximate the ideal trace. In this experiment, we show results only for aggressive optimization, which is gcc -o3, to avoid overestimating.

One approach to figure out the trace length is by creating an ideal environment where virtual registers are 32 and 64 respectively. As expected, Figure 6 shows performance speedup when the full execution trace is divided into a series of traces with a different number of basic blocks. On the surface, according to Figure 6, performance seems to have more improvement as the trace length is increased. However, the speedups become slow when traces are longer than 500 and 200 fragments respectively. Moreover, this phenomenon is consistent across the benchmarks and optimization levels. Clearly, the potential performance improvement of trace length is significant; the average improvement in speedup is 3 and 2*x*. Based on our results, we can conclude that long trace is a key to achieving good performance, and they deserve the further research in the future.

Conclusions

We have designed and implemented a distributed DBT–DistriBit to supply heterogeneous code translation services to thin clients in recent cloud computing environments. This solution factors code translation out of thin clients and locates it on a powerful server in cloud. With the help of cloud computing processing power and unlimited storage; thin client can focus on execution wholeheartedly. We also investigated the potential profit of different factors on program performance. Experimental results showed that the potential profits of the number of virtual registers and trace length are significant. From our study, researchers are better positioned to identify what is important for dynamic binary optimization.

ACKNOWLEDGEMENTS

Our research is supported by the National Natural Science Foundation of China (Grant No.60873209, 60970107, 60970108), the Science and Technology Commission of Shanghai Municipality (09510701600).

REFERENCES

Andres G, Brendan E, Mike S, David A, David M, Mohammad RH, Blake K, Graydon H, Boris Z, Jason O, Jesse R, Edwin S, Rick R, Michael B, Mason C, Michael F (2009). Trace-based just-in-time type specialization for dynamic languages. In ACM SIGPLAN Notices. 44: 465–478.

Andrés LC, Joseph AW, Adin S, Philip P, Stephen MR, Eyal DL, Michael B (2009). Snowflock: rapid virtual machine cloning for cloud computing. In Proceedings of the 4th ACM European conference on Computer systems, pp. 1-12.

Borin E, Youfeng W (2008). Characterization of dynamic binary translation overhead. PROCEEDINGS–AMAS-BT, pp. 4-13.

Brian F, Aqeel M, Francesso S, Sanjay JP, Steve SL (2004). The performance potential of trace-based dynamic optimization. In University of Illinois Technical Report, UILU-ENG-04-2208.

Chi KL, Robert C, Robert M, Harish P, Artur K, Geoff L, Steven W, Vijay JR, Kim H (2005). Pin: building customized program analysis tools with dynamic instrumentation. In ACM SIGPLAN Notices. 40: 90–200.

Cristina C, Mike VE (2000). UQBT: Adaptable binary translation at low cost. Computer. 33: 60–66.

Emin GS, Rober G, Arthur JG, Nathan A, Brian NB (1998). A. Gregory, and S. McDirmid, Distributed virtual machines: A system architecture for network computing. European SIGOPS, pp 13-26.

Emin GS, Robert G, Arthur JG, Brian NB (1999). Design and implementation of a distributed virtual machine for networked computers. In Proceedings of the seventeenth ACM symposium on Operating systems principles, ACM, pp. 202-216.

Haibing G, Yindong Y, Kai C, Yi G, Liang L, Ying C (2010). Distribit: a distributed dynamic binary translator system for thin client computing.

In HPDC-19, ACM, pp. 684–691.
Jim S, Ravi N (2005). Virtual machines: versatile platforms for systems and processes. Morgan Kaufmann Pub.
John SH, Gerald P (1994). File-system development with stackable layers. ACM Transactions on Computer Systems (TOCS). 12(1): 58–89.
Michael A, Armando F, Rean G, Anthony DJ, Randy HK, Andrew K, Gunho L, David AP, Ariel R, Ion S, Matei Z (2009). Above the Clouds: A Berkeley View of Cloud Computing. Technical Report No. UCB/EECS-2009-28.
Sorav B, Alex A (2008). Binary translation using peephole superoptimizers. In Proceedings of the 8th USENIX conference on Operating systems design and implementation, USENIX Association, pp. 177-192.
Vasanth B, Evelyn D, Sanjeev B (2000). Dynamo: a transparent dynamic optimization system. In ACM SIGPLAN Notices. 35: 1–12.
Vikram A, Chris L, Michael B, Anand S, Brian G (2003). LLVA: A Low-level Virtual Instruction Set Architecture. In ACM Micro-36, San Diego, California, pp. 201-216.
Yindong Y, Haibing G, Erzhou Z, Hongbo Y, Bo L (2010). Crossbit: A multi-sources and multi-targets dbt. In The First International Conference on Cloud Computing, GRIDs, and Virtualization, IARIA, pp. 41-47.