Cold Code Analysis

Wesley Attrot*
Unicamp

Guido Araújo
Unicamp

Abstract

Dynamic binary translators are programs that translate binary programs from one machine to another. The translation is done on the fly, so performance is a major issue in this kind of system. Identifying and optimizing hot traces is a way to achieve more performance, and also to compensate for the translation overhead. Aggressive optimizations need precise data/control-flow information about the code, otherwise they will be conservative and less effective. In this paper, we measure the amount of additional data-flow information one can obtain by going beyond hot trace boundaries into non-frequently executed (cold) code. We show that in some cases, as in liveness analysis, one can considerably improve the information available, thus creating more opportunities for trace optimization. Moreover, the amount of additional data-flow information decreases very fast as one departs from trace boundaries, limiting the overhead imposed by the cold code analysis.

1 Introduction

Dynamic binary translators (DBTs) are programs designed to execute in a host machine binaries from other machines, performing the translation on the fly. Alternatively, DBTs may be used to improve performance of binaries from the same architecture [4].

DBTs translate each instruction to be executed into a equivalent instruction in the host machine. Some DBTs have a codecache to store the translated instructions and to execute these instructions. The codecache optimization saves time, as it avoids the overhead of translating instructions repeatedly [4].

*This work was sponsored by CNPq and Intel

For optimization purpose DBTs can instrument the translated code so as to identify highly executed regions in the program and select them for analysis. Once optimized, these regions can improve the program runtime and hopefully hide the translation overhead.

Dynamic optimizations may include well-known optimizations like dead code elimination, copy propagation, CSE [2] and others that are more specific to the translation environment. Register allocation in particular, is an optimization that demands much more effort, particularly if the host and guest machines have very different register sets and sizes.

Like in off-line optimizers, on-the-fly optimizers need information about the code, which could be particularly hard to obtain in a dynamic environment. Dynamic environments create some new challenges and constraints to the optimizer. For example, it is not possible to construct the control flow graph (CFG) of the program, only program execution traces are available. Moreover, structures like variables and basic blocks are hidden behind the assembly code.

Off-line optimizers can spend a lot of time gathering information about the code and optimizing it, but dynamic optimizers need to be as fast as possible. The time spent optimizing the traces must pay itself in performance increase or the optimization will be useless. But even the fastest optimizer needs some kind of information about the code to do its job.

2 Trace Optimization

Traces are sequences of instructions, including branches but not including loops, that are executed for some input data [15]. If some trace is executed more times than a specified threshold we call it a

hot trace. DBT's hot traces are important for optimizations because their high execution frequency considerably improves optimizations' effectiveness, thus improving the overall program execution time.

```
(01) shl esi, cl
(02) add edx, ebx
(03) or eax, esi
(04) movzx eax, ax
(05) mov DWORD PTR [esp+01ch], eax
(06) mov ebx, DWORD PTR [edi*4+080cd380h]
(07) test ebx, ebx
(08) jnz ...
(09) mov eax, DWORD PTR [esp+024h]
(10) movzx ebp, WORD PTR [eax*2+0810dae0h]
(11) add eax, 0x1h
(12) mov DWORD PTR [esp+024h], eax
(13) cmp ebp, 0x100h
(14) jb ...
(15) mov eax, ebp
(16) shr eax, 0x7h
(17) movzx edi, BYTE PTR [eax+080d4560h]
(18) mov ebx, DWORD PTR [esp+020h]
(19) movzx ebx, WORD PTR [ebx+edi*4+02h]
(20) mov eax, ebx
(21) neg eax
(22) add eax, 0x10h
(23) cmp edx, eax
(24) jg ...
(25) mov eax, DWORD PTR [esp+01ch]
(26) movzx esi, WORD PTR [ebx+edi*4]
(27) jmp ...
```

Figure 1: Trace

Figure 1 shows a trace from the SPEC CPU 2000 [1] program 164.gzip in x86 assembly. Since this is a hot trace, we expect that the executing flow enters into the first instruction `shl esi,cl` and leaves out of the last instruction `jmp ...` most of the time. The execution flow may leave the trace in one of the 3 branch instructions in the middle of the trace, but this behavior is expected to occur few times. We call these branches of **side exits** because they break the continuous execution flow of the trace.

Assuming that this trace will execute most of the time without taking a side exit, we can focus the optimizations on this assumption. If we do not care about memory and register aliasing, a simple optimization that can be performed on the example trace is to remove a memory access. The 5^{th} instruction of the trace stores `eax` to `esp+01c` and the 25^{th} instruction on this trace reads this memory position. If we find an empty register covering these two instructions, we can copy the store value into the 5^{th} instruction to this register and replace the memory access at the 25^{th} instruction for a reference to a register. In fact, we notice that register `esi` is used at the 3^{rd} instruction and defined at the 26^{th} instruction. From the 5^{th} to the 25^{th} instruction there is no definition or use of `esi`, but we can not ensure that register `esi` is dead along these instructions.

There are 3 side exists between the instructions under analysis. The value of `esi` may be alive outside the trace on one of the side exits. If this is true, then the proposed optimization is not possible.

The common liveness analysis on this trace is not enough to solve this issue. For conservative reasons we must assume that all registers are alive on the side exists and at the end of the trace. This assumption turns the proposed optimization impossible.

3 Cold Code Analysis

Situations like the presented in Figure 1 restrict the aggressiveness of trace optimizations, allowing optimizations only inside basic blocks. If we always assume, for every optimization, the conservative way in every optimization, we will miss many optimization opportunities.

The proposed optimization to remove the memory access can not be performed under conservative estimation of registers' liveness. But if we inspect some instructions outside the trace, we can increase our information about the register liveness and maybe discover that `esi` is dead on all side exists, thus allowing us to do the optimization.

We call the code outside a trace **cold code**, because it is not supposed to execute as many times as traces, so is not profitable to optimize it. But on the other hand it has information that may be useful to the hot traces, like register and flags usage.

The analysis of cold code consists in performing the required data flow analysis on this piece of cold and gather the required information to increase the data flow analysis precision in the hot trace. Since

DBTs do not generate a full CFG, some heuristics must be employed to walk on the cold code.

Diving into cold code (or walk into cold code) has its own issues involved. How deep must we dive into cold code and how much time spent on it? We have no guaranties that this extra time spent on increasing data flow information will allow us to perform further optimization.

4 Diving into Cold Code

Figure 2 shows the initial configuration for a trace. This trace has 4 basic blocks and 3 side exits. This can be called level 0 cold code diving, that is, no cold code analysis. If we think about liveness analysis, we must assume that all registers are live at each side exit.

Figure 3 corresponds to dive one basic block into cold code (**level 1**). At this level only one basic block into cold code is analyzed. The end of the block is limited by a branch instruction, a function call instruction, a ret instruction or a jump to a non translated address. When we find one of these instructions, we assume that the registers that could not be solved are alive after this point.

If we decide to dive one more level (**level 2**), we must analyze the successors of basic blocks in level 1, when possible, as shown in Figure 4, where we can see that each basic block in level 1 has its successors added to the analysis.

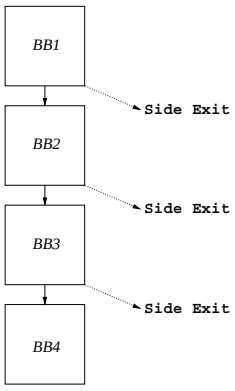

Figure 2: Trace for Optimization

We can expand the diving process to reach deeper levels, but it might not always be possible do dive so deep. Figure 5 shows a possible scenario.

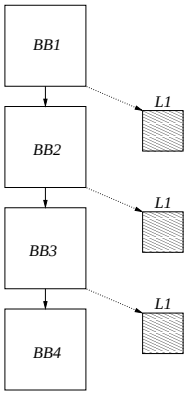

Figure 3: Diving one level into cold code

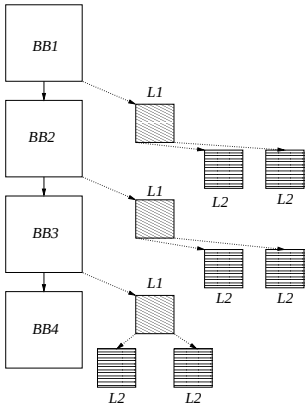

Figure 4: Diving two levels into cold code

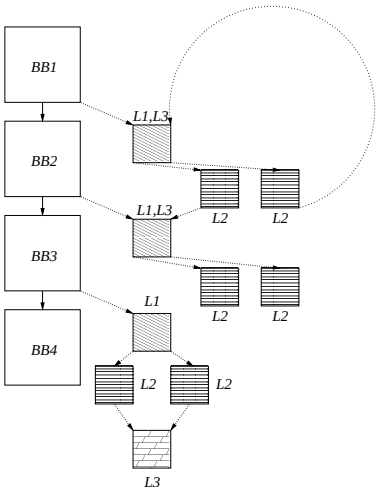

Figure 5: Diving three levels into cold code

If we reach a jump instruction that does not allow us to determine to target address, it is not possible to keep going with the analysis. The same happens with ret instructions, without looking at the stack. This put an extra cost to the cold code analysis.

In Figure 5 we can see that sometimes a deeper level can take us to a basic block at shallow levels. If we keep track of the information at each level entrance, we don't need to dive again, otherwise, we will be analyzing the same code again.

The blocks on the side exit of **BB1** go from level 2 to level 1 when trying to reach level 3. Sometimes they can reach the middle of a trace.

5 Experimental Results

Our experimental results are focused in the optimization proposed in the beginning of this paper, i.e. in finding available registers (dead registers) that can be used to replace memory accesses on a trace.

We implemented this technique into our DBT translating system, which performs code transformation from IA32 to IA32. Our DBT [6] runs on Linux and has its structure shown in Figure 6. It has a kernel module which is loaded and replaces the system call **execve**. Every time that a program executes, the module checks for a shared library in the same directory of the application, if this library is not found, then the original Linux execve is called. If the shared library is found, then it is loaded and starts the execution. This shared library is our DBT itself, which loads the application code and starts the translation process.

Our DBT has 3 main modules: *front-end*, *runtime* and *back-end*. The *front-end* module translates the application instructions and stores them into a code cache, controlling the program execution in this cache. The *runtime* module performs the communication between the DBT and OS, and between the application and the OS. To achieve that, it provides I/O interfaces, system calls, and support to system signals, dynamic shared objects loads and self-modifying code. The *front-end* and the *runtime* interact to handle system related features. The *front-end* is also responsible to select hot traces for runtime optimization by the *back-end*. The *back-end* module is responsible for trace optimization. It is able to dump hot-traces into files, so as to perform off-line optimization and profiling analysis.

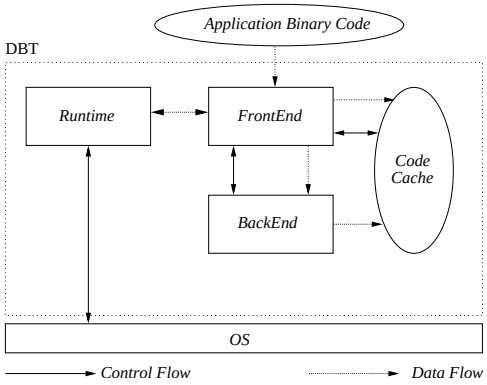

Figure 6: Our Dynamic Binary Translator Structure

5.1 Experiments

Our experiments used the SPEC CPU 2000 [1] benchmark. Programs were compiled with the Intel Compiler using the peak tuning and ref input. For each program the DBT identified hot traces, using MRET heuristics [4], and dove into cold code looking for available registers. The register set of x86 architecture creates some alias relationship among the registers. For example, a definition of register `EAX` also defines registers `AX`, `AH` and `AL`. A use of register `BH` implies in a use of register `EBX`. To handle these issues, we treat the registers as resources, and thus `EAX`, `AX` as well as the other registers are considered different resources.

In our experiments we analyze the set of registers presented in Table 1.

We analyzed every side exit of each trace constructed by our DBT. The set of experiments was performed using 8 dive levels. For the first run of SPEC, cold code analysis was done using just one dive level, a second run used two dive levels and so on, until we reach 8 dive levels. As shown in Table 1, at each side exit we are trying to determine the availability of 48 registers. The set of traces generated by our DBT produced a total of 240,612 side exits, so the maximum number of available registers is 11,549,376 registers.

In our implementation a basic block can be analyzed more than once because we do not keep track

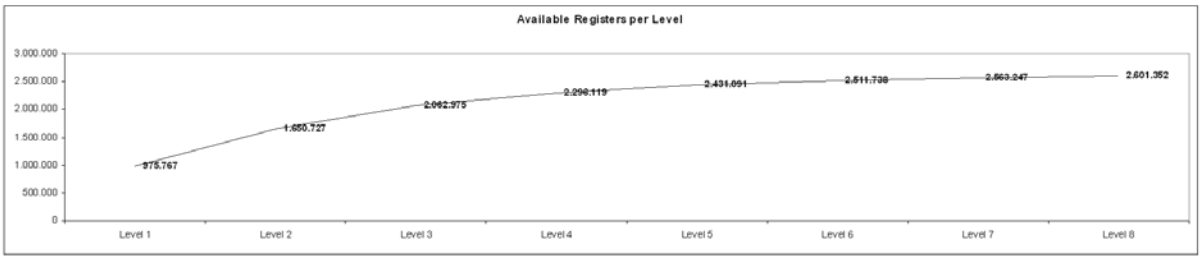

Figure 7: Registers available per Level

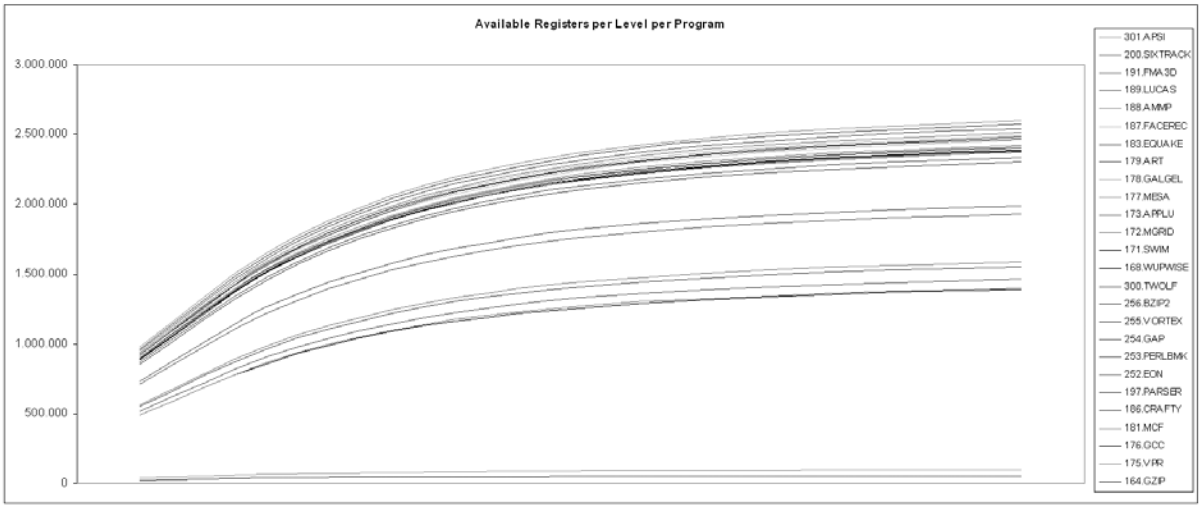

Figure 8: Registers available per Level per Program

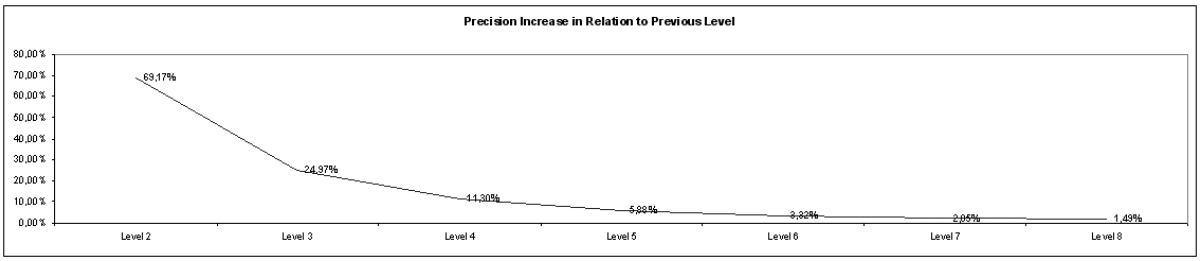

Figure 9: Increase of information accuracy

of the information collected at each block.

Figure 7 and Table 2 summarize the results that we found at each level. Diving 8 levels allowed us to discover that 22.52% of the registers are available (dead), that is, almost 1/4 of the whole register set. For an architecture with few registers, this could become an important resource which could be explored by a dynamic register allocator. Notice that we are discussing dead registers. If we try to determine unused live range intervals existing in live registers, this value could become larger than 22.52%.

By going beyond trace boundaries, cold code analysis can determine the live/dead status for almost all registers on a side exit, thus allowing the optimizer to make better choices of what to op-

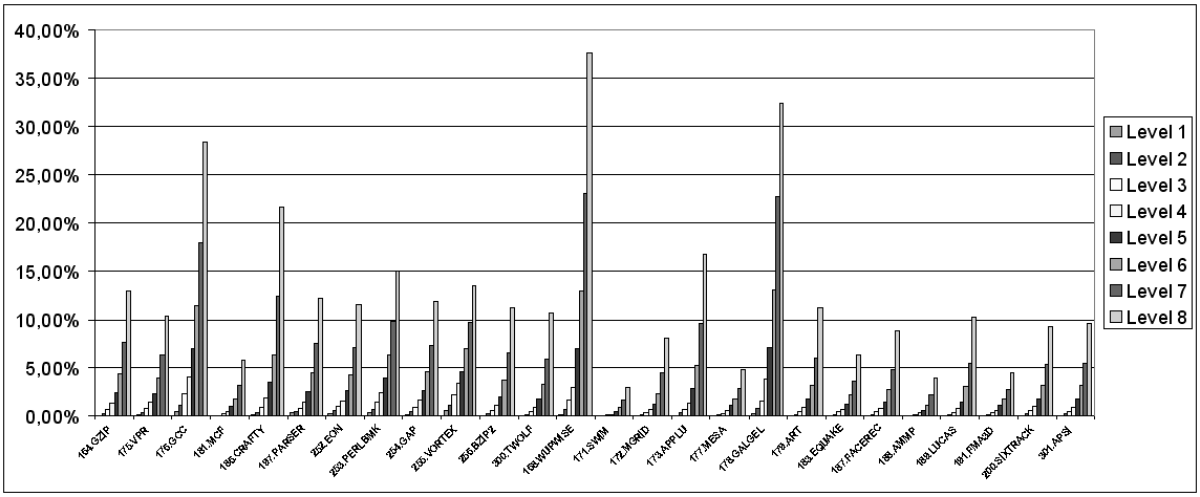

Figure 10: Cold Code Analysis Overhead

	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Level 7	Level 8
164.GZIP	0.11%	0.32%	0.75%	1.39%	2.48%	4.38%	7.63%	13.01%
175.VPR	0.17%	0.41%	0.86%	1.47%	2.34%	3.93%	6.42%	10.28%
176.GCC	0.49%	1.10%	2.31%	4.06%	6.91%	11.46%	17.96%	28.42%
181.MCF	0.09%	0.12%	0.27%	0.50%	1.00%	1.73%	3.23%	5.77%
186.CRAFTY	0.20%	0.41%	0.95%	1.88%	3.54%	6.33%	12.40%	21.62%
197.PARSER	0.40%	0.42%	0.83%	1.45%	2.57%	4.44%	7.45%	12.23%
252.EON	0.23%	0.58%	0.97%	1.59%	2.68%	4.30%	7.08%	11.51%
253.PERLBMK	0.34%	0.73%	1.43%	2.43%	3.94%	6.28%	9.76%	14.96%
254.GAP	0.17%	0.48%	0.93%	1.61%	2.66%	4.55%	7.27%	11.90%
255.VORTEX	0.51%	1.21%	2.17%	3.36%	4.69%	6.95%	9.70%	13.51%
256.BZIP2	0.14%	0.26%	0.61%	1.11%	2.02%	3.72%	6.58%	11.21%
300.TWOLF	0.11%	0.22%	0.47%	0.94%	1.78%	3.28%	5.97%	10.69%
168.WUPWISE	0.21%	0.76%	1.66%	3.05%	6.94%	13.00%	23.07%	37.59%
171.SWIM	0.01%	0.06%	0.14%	0.19%	0.47%	0.93%	1.67%	3.03%
172.MGRID	0.05%	0.16%	0.34%	0.69%	1.25%	2.40%	4.45%	8.05%
173.APPLU	0.13%	0.32%	0.69%	1.41%	2.84%	5.28%	9.57%	16.70%
177.MESA	0.05%	0.17%	0.24%	0.51%	1.07%	1.74%	2.91%	4.85%
178.GALGEL	0.29%	0.78%	1.58%	3.81%	7.15%	13.03%	22.76%	32.45%
179.ART	0.08%	0.16%	0.45%	0.92%	1.69%	3.19%	6.00%	11.23%
183.EQUAKE	0.07%	0.16%	0.44%	0.71%	1.26%	2.15%	3.65%	6.26%
187.FACEREC	0.04%	0.21%	0.42%	0.83%	1.48%	2.74%	4.86%	8.88%
188.AMMP	0.05%	0.10%	0.20%	0.36%	0.65%	1.22%	2.17%	3.88%
189.LUCAS	0.06%	0.19%	0.37%	0.83%	1.50%	3.09%	5.47%	10.27%
191.FMA3D	0.08%	0.19%	0.39%	0.67%	1.07%	1.73%	2.78%	4.47%
200.SIXTRACK	0.10%	0.23%	0.55%	1.01%	1.76%	3.16%	5.42%	9.33%
301.APSI	0.10%	0.24%	0.50%	0.94%	1.80%	3.18%	5.51%	9.57%

Table 4: Cold Code Analysis Overhead

Registers in x86 Architecture
EAX AX AH AL
EBX BX BH BL
ECX CX CH CL
EDX DX DH DL
EBP BP
ESI SI
EDI DI
ESP SP
MM0 MM1 MM2 MM3 MM4 MM5 MM6 MM7
XMM0 XMM1 XMM2 XMM3 XMM4 XMM5 XMM6 XMM7 XMM8 XMM9 XMM10 XMM11 XMM12 XMM13 XMM14 XMM15

Table 1: x86 registers

Available Registers	
Level 1	975,767
Level 2	1,650,727
Level 3	2,062,975
Level 4	2,296,119
Level 5	2,431,091
Level 6	2,511,738
Level 7	2,563,247
Level 8	2,601,352

Table 2: Available registers per level

Precision Increase	
Level 1	–
Level 2	69.17%
Level 3	24.97%
Level 4	11.30%
Level 5	5.88%
Level 6	3.32%
Level 7	2.05%
Level 8	1.49%

Table 3: Precision increase per level

timize or not. Notice that the registers that are considered available, are those that we can ensure that were assigned a value on the side exit. If a register never is used in a program, so this register will be considered as not available, because we will never find a instruction defining it.

Figure 9 and Table 3 give us an idea on how deep the analysis should dive. The percentage numbers are related to the previous level. From level 1 to level 2 the precision increase in the number of available registers was 69.17%, while from level 2 to level 3 the precision increase was 24.97%. This diminishing return behavior continues until we reach the last level (level 8). As shown, after level 4 or 5 the increase in the number of available registers is not large, meaning that for most of the purposes, diving until level 4 or 5 is enough to determine the dead/live status of most registers.

Table 4 and Figure 10 show the time execution overhead caused by our cold code analysis. The overhead corresponds to the amount of time spent by the analysis, with respect to the whole program execution time. As shown, until level 3 the overhead is below 1%, in 21 out of the 26 programs available. As expected, Figure 10 shows that the cost is exponential, and thus the overhead for the final levels is very high. This probably happens because some basic block may be analyzed several times, as for example, in the case of a loop. An implementation which considers code re-analysis could certainly decrease this overhead.

6 Conclusion

In this paper we examined a way of increasing data flow information available on traces constructed by DBTs. The process consists in analyzing cold code, beyond program hot traces, to obtain the desired information. As far as we know, no DBT employs this technique to analyse code outside traces.

We implemented a data flow analysis to determine the set of available registers at each side exit of a trace. To increase the precision of the data flow information, we performed cold code analysis, running the benchmark several times and analyzing a large amount of cold code at each run.

The tests were performed using the whole SPEC CPU 2000 benchmark and the results allowed us to conclude that almost 1/4 of the x86 architecture

registers are dead on trace exits. Moreover, the results also showed that there is no need to go far into cold code in order to achieve this information: a level 4/5 dive is enough.

Regarding the example presented in Figure 1, we were able to determine that register `esi` was dead on the side exits at instructions 8, 14, 24 enabling us to perform the optimization.

For the future we plan to change our code analysis technique so as to avoid the overhead of repeating basic block analysis. We also intend to test it in other types of data flow analysis, like for example register flag detection, which is a very relevant information in other code optimizations.

References

[1] *Standard Performance Evaluation Corporation (SPEC)*. http://www.spec.org.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[3] E.R. Altman, D. Kaeli, and Y. Sheffer. Welcome to the opportunities of binary translation. *IEEE Computer*, 33(3):40–45, Mar 2000.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. *SIGPLAN Not.*, 35(5):1–12, 2000.

[5] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigal Zemach. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium-based systems, 2003.

[6] Edson Borin, Cheng Wang, Youfeng Wu, and Guido Araujo. Software-based transparent and comprehensive control-flow error detection. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 333–345, Washington, DC, USA, 2006. IEEE Computer Society.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35(11):202–211, 2000.

[9] Kemal Ebcioglu, Erik Altman, Michael Gschwind, and Sumedh Sathaye. Dynamic binary translation and optimization. *IEEE Trans. Comput.*, 50(6):529–548, 2001.

[10] Kemal Ebcioglu and Erik R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[11] David Hiniker, Kim Hazelwood, and Michael D. Smith. Improving region selection in dynamic optimization systems. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 141–154, Washington, DC, USA, 2005. IEEE Computer Society.

[12] J.D. Hiser, N. Kumar, Min Zhao, Shukang Zhou, B.R. Childers, J.W. Davidson, and M.L. Soffa. Techniques and tools for dynamic optimization. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 8 pp.–, 25-29 April 2006.

[13] Bich C. Le. An out-of-order execution technique for runtime binary translators. *SIGOPS Oper. Syst. Rev.*, 32(5):151–158, 1998.

[14] J. Lu, H. Chen, P. Yew, and W. Hsu. Design and implementation of a lightweight dynamic optimization system, 2004.

[15] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 340 Pine Street, Sixth Floor, San Francisco, CA 94104-3205, USA, 1997.

[16] M. Probst, A. Krall, and B. Scholz. Register liveness analysis for optimizing dynamic binary translation. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 35, Washington, DC, USA, 2002. IEEE Computer Society.