Dynamic Register Promotion of Stack Variables

Jianjun Li and Chenggang Wu *

Key Laboratory of Computer System and Architecture
Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China
{lijianjun,wucg}@ict.ac.cn

Wei-Chung Hsu

Department of Computer Science
National Chiao Tung University, Hsinchu, Taiwan
hsu@cs.nctu.edu.tw

Abstract

Dynamic Binary Translation (DBT) has been widely used in various applications. Although new architectures and micro-architectures often create performance opportunities for programmers and compilers, such performance opportunities may not be exploited by legacy executables. For example, the additional general purpose and XMM registers in the Intel64 architecture do not benefit the IA-32 binaries. In this paper, we designed and developed a DBT system to dynamically promote stack variables in the source binaries to the additional registers of the target architecture. One of the most challenging problems is how to deal with the possible but rare memory aliases between promoted stack variables and other implicit memory references. We devised a runtime alias detection approach based on the page protection mechanism in Linux and a novel stack switching method to catch memory aliases at run-time. This approach is much less expensive than traditional approaches like inserting address checking instructions. On an Intel64 platform, our DBT system with speculative stack variable promotion has sped up several SPEC CPU2006 benchmarks in IA-32 code, with the largest performance gain over 45%.

Categories and Subject Descriptors D.3.4 [*Processors*]: Code generation, Optimization, Run-time environments

General Terms Performance, Experimentation

Keywords register promotion, alias detection, dynamic optimization

1. Introduction

Dynamic Binary Translation (DBT) has been widely used in various applications [1, 3]. For fast emulations and process virtual machines [24], dynamic binary translation is almost a standard technique to use. In process virtual machines, when the target Instruction Set Architecture (ISA) is identical to the source ISA, DBT can be applied to dynamic binary optimization [16, 17], dynamic instrumentation [4, 20], and dynamic software security enforcement [10].

For DBT systems with incompatible ISA, it is quite common that the host architecture has more registers than the guest architecture. For example, in IA-32 EL [2], the host Itanium has 128 GPRs and in FX!32 [5], the host Alpha has 32 registers, far more than the 8 registers in IA-32. Additional registers can be used for more effective code scheduling and for promoting variables to registers. In the same ISA family, computer companies have kept introducing new architecture features and micro-architectures to increase functionality as well as performance. Consider the Intel64 (i.e. the x86-64 ISA), for example, it provides additional 8 general purpose registers (GPR) and 8 XMM registers, and wider data paths registers than IA-32 [6, 7]. While recompiling can often take good advantages of such new features, in practice, a large number of application binaries running on user sites are legacy code when new generation processors are born. Dynamic binary optimization enables legacy binaries to exploit such architecture features. However, existing dynamic binary optimizers [1, 3, 17] often focus more on exploiting applications' dynamic behavior in optimizations such as procedure in-lining and cache prefetching, instead of exploiting new architecture features. Different from prior research, this paper focus more on how to promote more variables to the additional registers during binary translation and optimization process.

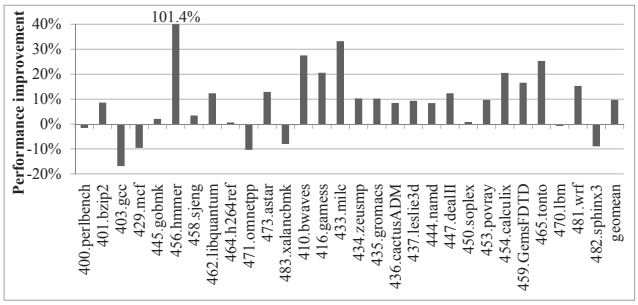


Figure 1. Performance comparison of Intel64 code and IA-32 code. (Y-axis shows the performance improvement of Intel64 code over IA-32 code running on Intel E5520.)

To get a feeling about such a performance opportunity, we compile the SPEC CPU2006 benchmarks to IA-32 and Intel64 code, respectively, and run them on the same Intel Nehalem based machine. Figure 1 shows that the 64-bit code outperforms IA-32 code significantly for most of the benchmarks (with a geometric mean of 10% greater performance). However, there are a few benchmarks showing performance degradation, mostly due to the code and data size increase for the 64-bit architecture. Overall, many more benchmarks benefit from the additional registers. Unfortunately, this performance opportunity is currently not available to legacy (i.e. IA-32) binaries.

This paper studies how to effectively utilize the additional registers of target architectures to improve the performance of dynami-

* To whom correspondence should be addressed.

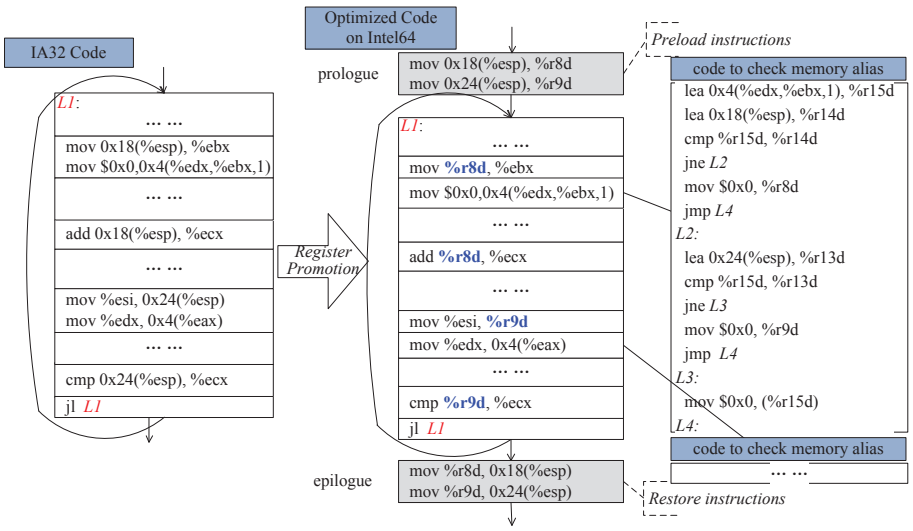


Figure 2. A simple example of register promotion for stack variables

cally translated code. In the context of DBT, promoting variables to registers is known to be challenging since alias analysis on executables is far more constrained than in source code compiling [9, 25]. Many proposed approaches [13, 15, 21, 22] either require hardware support, or incur excessive overhead in generated code, and may not be adopted in a practical DBT system. We have been limiting the register promotion optimization to stack variables. In most binaries, stack variable references can be divided into either explicit where the base address is the stack/frame pointer registers (e.g. %esp/%ebp in IA-32 code), or implicit where the base address is some registers other than stack/frame pointer registers. Implicit references are often pointer based. It is straightforward to distinguish explicit stack references, and some of the frequently used ones are candidates for register promotion. However, the implicit references are hard to analyze, and they could alias to any explicit references. We have devised a set of engineering solutions based on virtual page protection and stack switching mechanisms to detect such rare, but possible, exceptional cases at run-time. If there is a true alias exists between an implicit memory reference and a promoted stack variable, the page protection mechanism will generate a trap and our installed exception handler will undo the optimization. Since the true aliasing is rare, this speculative optimization is very effective in practice. Although our engineered solutions are designed for stack variable promotion, they may also enable more aggressive code scheduling because there is a safety net to catch violation of memory dependencies (on stack variables) at runtime.

We have designed and developed a DBT system to dynamically promote stack variables to registers. This DBT system can dynamically promote stack variables of IA-32 binaries to the additional registers in Intel64. In Intel64, the cost of moving data between the floating point registers and the integer registers has been significantly reduced, so it becomes profitable to keep frequently used integer variables in idle floating point registers. Our DBT system uses the additional integer, floating point, and XMM registers for promoting stack variables. On the SPEC2006 benchmark, our DBT system has increased the performance of some benchmarks by as much as 45%.

This work makes the following contributions:

- It proposes a memory alias detection method for stack variables. The method is based on virtual page protection and dynamic stack switching. The proposed scheme can enable aggressive speculative optimizations such as register promotion in DBT systems.
- It implements the proposed scheme in a DBT for the Intel64 machines to promote stack variables to registers. Several SPEC CPU2006 benchmarks have been sped up with the largest performance gain over 45%.
- It provides technical and engineering details of the implementation of the DBT including practical problems encountered with OS and dynamic profiling.
- It gives in-depth performance analysis to verify the correctness and validate the performance gain of our DBT system.

The rest of this paper is organized as follows. Section 2 describes our proposed solutions to the challenge of dynamic register promotion. Section 3 presents a detailed implementation of our DBT system on the Intel64 platform. Section 4 provides the experimental setup and Section 5 discusses the experimental results. Section 6 gives a brief overview of related work. Section 7 summarizes and concludes this work.

2. Dynamic Register Promotion of Stack Variables

Promoting variables to registers have long been an important compiler optimization [18, 19]. On modern microprocessors, the optimization is even more important since memory instructions are relatively more expensive than ALU operations in superscalar or VLIW processors. In the X86 architecture, if an operand can be directly accessed from a register instead of the memory, the instruction used could be shorter, and fewer micro-operations would be generated from the macro-instruction at execution time on Pentium Pro and later micro-architectures.

We attempt to promote more stack variables in legacy code to registers during dynamic binary translation. Although variables other than stack variables can also benefit from register promotion, it is rather difficult to obtain sufficient information from the executable to do it safely. In this section, we use the IA-32/Intel64 [6, 7] as an example platform to illustrate our techniques.

2.1 Explicit and Implicit References of Stack variables

Stack variables in IA-32 binaries are usually referenced either explicitly or implicitly. Explicit references are those memory references with addressing mode where the base address register is %esp (stack top) or %ebp (frame pointer). Implicit references are those memory references with addressing mode where the base address register is something other than %esp or %ebp[1]. Pointer dereferencing is implicit reference since the pointer is stored in a register other than the stack top or the frame pointer.

It is relatively easy to disambiguate explicit references. For example, we can tell that *0x4(%esp)* is referencing a different variable than *0x8(%esp)* in the IA-32 binary if the size specified is a word. It is much more different to do so for implicit references. For example, it is hard to tell that *0x4(%eax)* is referencing a different memory location than, or has no overlapped data access with, *0x8(%ebx)*.

2.2 Example of Stack Variable Promotion

Figure 2 shows a simple example of register promotion of stack variables. An original IA-32 code segment is given on the left hand side. The code in this example accesses two stack variables, *0x18(%esp)* and *0x24(%esp)*, in a loop. Suppose there is no aliasing between the two stack variables and other memory references in the loop, we can safely promote the two variables into the extra registers on the Intel64 machine. The optimized code is given on the right hand side of the arrow. A prologue and an epilogue have been added in the optimized code. The prologue loads the two stack variables into register %r8d and %r9 (%r8 and %r9 are newly added registers on Intel64 architecture), respectively, before entering the loop. Hence, the instructions in the optimized loop can directly use registers for the two variables. Four memory reference instructions in the loop body are now replaced by register-register instructions. At the exit of the loop, the epilogue uses two store instructions to put data back to memory because there may have operations accessing the two variables outside the optimized loop. In the optimized code, four memory references are eliminated and the code becomes smaller, so the overall execution time will likely be reduced.

2.3 Outstanding Issues for Dynamic Register Promotion

Three important issues must be addressed before applying dynamic register promotion. The first issue is how to ensure no possible memory aliases exist between promoted variables and other memory references. What if there is one memory reference to the promoted variable via a pointer stored in a register. If we do not have a way to catch such possible aliases, the above transformation is illegal. The second issue is what scope for applying register promotion - should it be a function, a loop, a region, or a basic block. Since the prologue/epilogue overhead may be too high if we target a basic block, we should target at a larger granularity like a loop or a function. Our DBT system must collect runtime information to determine what would be an ideal unit for optimization. The third issue is how to select candidates when promotable variables are more than the registers available. Should we favor the variables with a large reference count or the variables that are on the critical path? We discuss these issues in detail in the following subsections.

2.3.1 Memory Alias Detection

The optimization in Figure 2 is based on an optimistic assumption that no aliases exist between promoted variables and other memory references. However, the assumption is not always true. Therefore, we need a mechanism to handle the cases when the assumption does not hold at runtime. One simple method is adding address checking instructions to guard for exceptions. For example, the code in Figure 2 has two pointer based memory accesses, *0x4(%edx,%ebx,1)* and *0x4(%eax)*, which may alias to the promoted stack variables. The checking code is shown on the right hand side of Figure 2. When aliases are detected, the operation will be redirected to the mapped register. However, the cost of this method will increase very fast when multiple stack variables are promoted to registers, therefore, this approach is not viable due to its excessive overhead in practice.

Since it is generally recognized that aliases between pointers and stack variables are rare, we could use more efficient methods. In [14], speculative register promotion is performed with help from special hardware supports such as the ALAT (Advanced Load Address Table) to minimize the cost of address checking. However, ALAT is a specific feature in the Itanium processor family, not available in other architectures. Therefore, we need a more general and efficient method that can be applied to all architectures. The basic idea is to use the general page protection mechanism to detect true memory aliases between any memory references and the promoted stack variables.

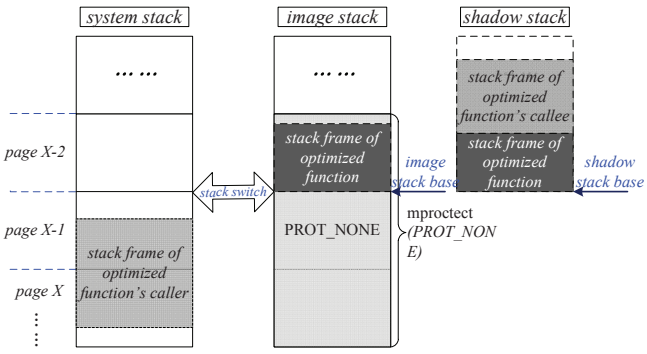


Figure 3. Using page protection mechanism to detect memory aliasing at runtime

Problems with a Simple Virtual Page Protection Method After promoting stack variables to registers, we use the system call *mprotect* to set the protection status of the virtual pages which contain promoted variables to *PROT_NONE* (as shown in Figure 3). This will cause any access to the protected virtual pages to trap. In our installed trap handler, we can check if there exists an alias between the promoted stack variables and the offending memory reference. However, this simple page protection method has two new problems. One is a large number of unwanted traps to the protected virtual page may arise when the program accesses non-promoted stack variables (Problem P1 - traps from non-promoted stack variables). The other problem is that the OS kernel may access the protected page when it puts the arguments of the trap handler onto the user stack. In this case, the program will abort (Problem P2 - traps from the OS).

A Shadow Stack to Avoid Unwanted Traps Ideally, we would like to promote all stack variables to registers so that no explicit stack references remain. This is not feasible in practice – some stack variables will not be promoted due to insufficient registers and some other reasons, and their references will touch the protected virtual page and cause traps. So to avoid unwanted traps from non-promoted stack variables, we should redirect the non-promoted stack variables to a different virtual page without special protections. One way is to allocate a large number of virtual registers in

[1] In some binaries, the frame pointer register (i.e. %ebp) may be used as general purpose registers. To correctly identify all stack variables, we conduct a simple data flow analysis to find out whether %ebp is used as a general purpose register.

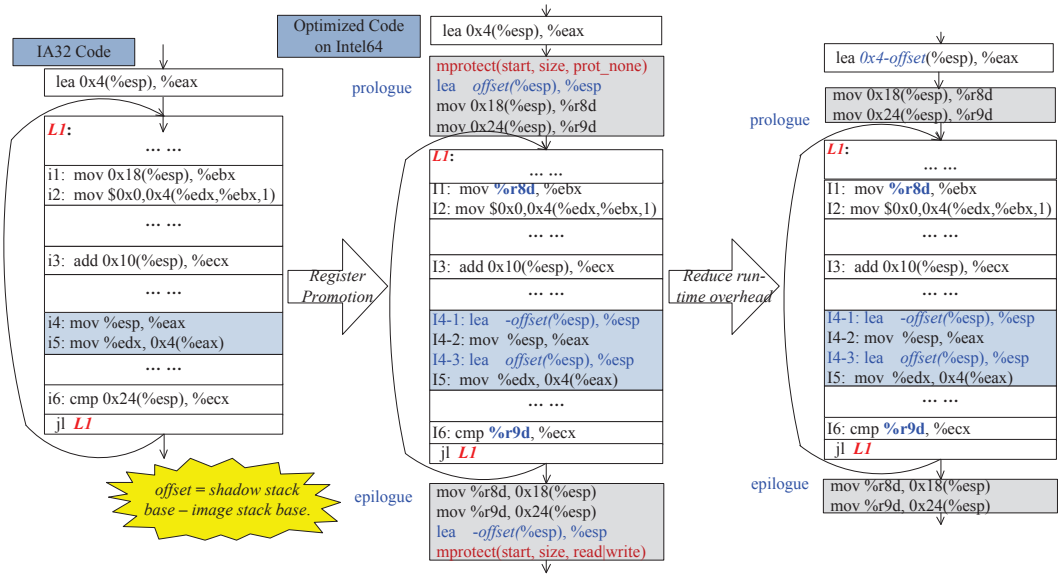


Figure 4. Example of optimized code using a page protection mechanism to detect memory alias *(In the figure, instruction "lea offset(%esp), %esp" is used to switch the stack pointer to the shadow stack, and instruction "lea -offset(%esp), %esp" is used to switch the stack pointer back to the image stack. In the optimized code, 0x10(%esp) is not promoted to register, and instruction I3 will access the shadow stack.)*

memory, so that such variables will be referenced to the page for virtual registers. However, this will require many extra data transfer operations to move data around. In this paper, we adopted the alias mapping mechanism [26] which maps multiple virtual pages to the same physical page while assigning different protections for each virtual page. Because the original system stack is not allowed to be alias mapped, we first build an image stack at the beginning of the DBT system. When the application enters an optimized function (a function with stack variables promoted to registers by our DBT), the stack pointer will switch to the image stack, and the arguments of the optimized function are copied to the image stack. Instructions to switch the stack frame between the system stack and the image stack are inserted at the entry and exit of the optimized function.

To prevent the accesses to the image stack out of range, the protection status of several high-address virtual pages of the image stack are also set to *PROT_NONE*. We then create a shadow stack, which is a set of different virtual pages, mapped to the same physical pages of the image stack. On Linux, this can be done by calling *mremap(old_address, old_size, new_size, flags)*, with *old_size* setting to 0. In the optimized function, we replace all instructions that explicitly reference the non-promoted stack variables with instructions that reference the corresponding locations in the shadow stack. This would force all explicit stack references of the non-promoted stack variables to access the shadow stack, and unwanted traps can be avoided.

Stack Switching Between the Protected Image Stack and the Unprotected Shadow Stack In an optimized function, the stack pointer should point to the top of the image stack so that all implicit references that touch the stack will get protection faults. However, those explicit stack references from non-promoted stack variables should access the shadow stack to avoid traps. Therefore, the stack pointer must be switched between the image stack and the shadow stack.

To address the problem P2 (traps from the OS), we need the stack pointer points to the shadow stack, which is not protected. However, we must ensure all implicit references to the stack accessing the image stack so that aliases can be detected. To achieve this purpose, stack switching between the image stack and the shadow stack must be carefully managed.

The key is to make sure all pointers to the stack in the optimized function are created with the address of the image stack. In the optimized function, we make the stack pointer point to the image stack before any instructions that moves or generates %esp or %ebp related addresses to create a pointer. So any pointers to the stack are created with addresses from the image stack. The stack pointer will then be switched to the shadow stack after those instructions (as shown in the middle of Figure 4). In this way, all implicit references to the stack will be referred to the image stack for alias checking. If the optimized code contains function calls, the callee's stack frame will be on the shadow stack, but when the callee accesses the optimized function's stack frame by any pointer arguments, those references will refer to the image stack. Any true aliases to the stack will be caught at run-time.

With this scheme, all explicit stack references to the non-promoted stack variables are redirected to the shadow stack which will not cause traps. All implicit references (pointer based) to the optimized function's stack frame will cause a trap (with the *SIGSEGV* signal). The physical memory usage is nearly the same as in the original program because both the image stack and the shadow stack are mapped to the same physical memory (except for minor differences potentially caused by the stack switch mechanism and the run-time system).

Minimizing the mprotect calling overhead As shown in Figure 4, there is a *mprotect* system call in the prologue of the optimized function. This system call sets protection for the image stack. Since a system call is expensive, this *mprotect* call could incur excessive overhead if the optimized function is called frequently. However, since the image stack is always set to *PROT_NONE*, we could insert the *mprotect* call in the initialization of the DBT system. At the entry of each optimized function, we can switch the stack pointer to the shadow stack directly. To ensure all implicit stack references go to the image stack, we switch the stack pointer to the image stack before any instructions which use the %esp or %ebp registers to create pointers (as shown in the right hand side of Figure 4). With this optimization, the cost of *mprotect* is paid only once.

2.3.2 Trap Handling

Our DBT system registers a trap handler to take the *SIGSEGV* signal. In the trap handler, the following steps are taken:

1. It first obtains the memory address that caused access violation from the context of the trap point.
2. If the trap address is not in the address range of protected pages in the image stack, the trap is not caused by our memory alias detection mechanism. In this case, the same signal is re-delivered to the program's default trap handler.
3. If the instruction which caused the trap is from the optimized function's callee, the optimized code will be discarded.
4. For traps occurred in the optimized code, we try to generate alias checking instructions for that offending instruction. To minimize alias checking instructions, we partition the address range of current stack frame into two parts. One part contains all promoted stack variables (we call it aliased area in this paper) and the other part contains all others. In this way, we can simply check if the trap address is outside the aliased area. If it is, alias checking instructions will be patched in. If the trap address is within the aliased area, the optimization of current function must be abandoned. In such a case, the value of promoted stack variables is restored to memory and the control is returned to the original code.

   We patch the offending instruction with alias checking instructions so that it will not cause a trap in the future. This is our way to avoid recurring traps. Another way is to simply discard optimized code and back to the original code. However, this approach seems too conservative because the memory references of this instruction may be always outside the aliased area. Since the patched instructions would incur run-time overhead, the optimized code will be discarded if the number of instructions trapped exceeds a threshold.

2.3.3 Granularities of Register Promotion

To minimize the prologue and epilogue overhead, we only consider function and loop as our optimization unit. Consider a function as the unit, for example, the epilogue at the end of the function may be skipped if we promote only stack variables. Besides, for function which contains multiple hot loops, optimizing the whole function can reduce the overhead of preload and post-store instructions. Nevertheless, the instructions outside the loops could incur more run-time overhead. For example, function calls outside the loop may run into extra traps since the callee may access the protected pages.

On the other hand, if we apply register promotion only to hot loops, we avoid the downside mentioned before, but may need to pay overhead of prologue and epilogue for each loop. A better compromise would be selecting the target based on the number of loops inside a function, the number of function calls outside loops, and the number of instructions outside loops. In our current implementation, if there are no functions calls outside loops, and the function contains more than three hot loops or there are a small number of instructions outside loops, we will select the whole function as the optimization target. Otherwise, we will select each hot loop as our optimization target.

2.3.4 Priority based Promotion

In many cases, there are more stack variables than available registers in a function or a loop. Therefore, the stack variables within the optimization unit should be prioritized for promotion.

One straightforward method is to sort the candidates according to their memory access type, their reference frequency and the loop nesting level of the instruction that references the stack variable. One variation is to distinguish loads from stores. Since loads are more exposed to access latency, we could give loads a higher priority than stores. So a stack variable occurs in multiple loads are more likely to get promoted to register.

However, we have observed that in many cases, the load latency can be hidden by dynamic instruction execution. It would be more effective if we promote those stack variables which are referenced on the critical paths. The latency of operations on the critical path is more difficult to hide in out-of-order superscalar processors. Therefore, in addition to the previous criteria of using the number of load operations and the loop nesting level, we will also give stack variables which are used on the critical paths a higher priority.

3. Implementation of Dynamic Register Promotion

To evaluate the mechanism proposed in section 2 in a DBT system, we choose IA-32/Intel64 as the source/target architecture. We attempt to use the newly added registers in the Intel64 architecture for promoting stacks at run-time.

3.1 Background

We choose the Intel Xeon E5520 processor [6] as the target for our experimental system. The E5520 based system supports the basic execution environment of IA-32 processor, and a similar environment under the IA-32e mode to execute 64-bit programs (in 64-bit sub-mode) and 32-bit programs (in compatibility sub-mode). The compatibility mode allows most legacy 16-bit and 32-bit applications to run directly under a 64-bit operating system. The Intel64 architecture has 16 more registers (8 general purpose and 8 XMM registers) than the IA-32 architecture, but those registers are only available in 64-bit sub mode, which does not run 32-bit binaries. So when running 32-bit applications on the Intel64 processors, half of the registers are wasted [6].

In order to accelerate 32-bit applications with the 16 additional registers in the Intel64 processors, we need to switch between 32-bit code (running in compatibility mode) and 64-bit code (running in 64-bit mode). Furthermore, we need a dynamic binary translator to convert IA-32 code into Intel64 code for the time consuming functions. We discuss the mode switching and dynamic binary translation in the following subsections.

3.2 Mode Switching between IA-32 and Intel64 code

In Intel64 processors, we can switch sub-modes by changing the value of the control bits in the code segment descriptor. In user mode, we cannot modify the code segment descriptor directly. However, this can be achieved indirectly by selecting an alternative entry of the Global Descriptor Table (GDT) in Linux. Since the GDT table is indexed by the CS register, we can switch operating mode by changing the value of the CS register. In user space, there are several ways to change the value of the CS register, for example, we may use far call (lcall), far ret (lret) or far jump (ljmp) instructions to do the job.

3.3 IA-32 to Intel64 Translation Issues

Some IA-32 instructions are not legal when running in 64-bit mode. Such instructions must be translated into Intel64 instructions when we switch to the 64-bit mode. For example, the single-byte-opcode instruction INC/DEC is not available in the 64-bit mode, which must be replaced by other equivalent instructions. Another example is that the push r/m32 instruction does not exist in the 64-bit mode, so it must be emulated with lea and mov instructions.

Address calculation from 32-bit code to 64-bit code must be handled with care. The default address size is 64 bit in the 64-bit mode, so all memory reference addresses are extended to 64-bit. In

most cases, extending the legacy instruction's 32 bit address to 64 bit will have no problems because the higher 32 bits of all GPRs are zero. However, if the addressing mode is SIB (scale-index-base) and the value of index register is negative, the calculated address may fall out of the 32-bit address range (this will cause an access violation at run-time). Although the default address size can be overridden by an address-size prefix, this would incur significant runtime overhead. In the 64-bit mode, if we override the address size to 32 bit, the address calculation process will require more cycles.

To avoid performance degradation, we add the address-size prefix only when it is necessary. We do not insert address-size prefixes during the 32-bit to 64-bit code translation. However, when some of such instructions trap at runtime due to access out-of-range, our trap handler may patch the offending instructions with address-size prefix inserted. If many of such cases occurred at runtime, a re-translation is called for to avoid further traps.

3.4 Register Allocation and Instruction Selection

There are 8 extra GPRs and 8 extra XMM registers available in the 64-bit mode, but the GPR and XMM registers cannot be used together in the same ALU instruction. For example, we cannot promote the stack variable of instruction add 0x4(%esp), %eax to an XMM register. To take full advantage of the extra registers, we divide instructions into three groups:

- GPR-only: The stack variables used in this group of instructions should be promoted to GPRs.
- XMM-only: The stack variables used in this group should be promoted to XMM registers.
- All: For data transfer instructions, we can use the extra registers as fast spill destination. In this case, both GPR and XMM registers are allowed. Besides, if the MMX registers are not used in the optimized function, they can also be used to promote stack variables.

If a stack variable is used in both GPR-only and XMM-only instructions, this variable will not be promoted to a register.

3.5 Target Function Selection

We use all extra registers as caller-saved registers, so we must save the value of promoted variables before a function call and load them back after the call. Furthermore, if there are pointer arguments in a function call, any reference to that argument in the callee will touch the protected image stack and cause an access violation trap. To minimize such overhead, we apply our register promotion optimization only to hot functions rather than all functions. However, if a function is hot because a high calling frequency, not because of a large invocation time, it may not be a good candidate because the prologue/epilogue overhead could exceed the stack variable promotion benefit. In our implementation, we consider the invocation time (in milliseconds/call) as one primary criterion in selecting hot functions.

Most modern processors provide Hardware Performance Monitoring (HPM) to assist performance monitoring and profiling. In our work, we take advantage of the Performance Monitoring Unit (PMU) on the Intel Xeon E5520 processor to locate hot spots of the program [11]. We use the libpfm4 library in Linux to access the performance counters. To find the hot spots in the program, we take a sample every 266K unhalted CPU cycles. After collecting samples for a second, we analyze the samples to identify hot functions (any function with more than 10% of the total samples), and the identified hot function is instrumented to collect the number of invocations. After instrumentation, we continue sampling the program for another second. Hot functions with a relatively large invocation time (greater than 0.001ms) will be selected as the optimization target. After the optimization targets are selected, we remove instrumentation and stop profiling. The selected functions are optimized by the helper thread (also called the optimization thread) while the original program is still running. To adapt to phase changes in the program, we restart sampling and profiling every 12 seconds. If a large percentage of samples are not within the optimized code, optimization process will be invoked again.

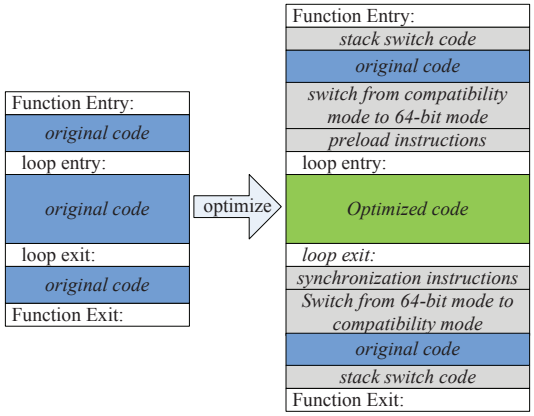


Figure 5. Overview of optimized code

Table 1. System Configuration

CPU	4-way, 2.27GHz Intel Xeon E5520 (Nehalem)
L1 ICache	32KB/core, 4-way set associative
L1 DCache	32KB/core, 8-way set associative
L2 Cache	256KB/core, 8-way set associative
L3 Cache	8MB/4 cores, 16-way set associative
Memory Size	8 GB
Operating System	Redhat Enterprise Linux 5.4 (64-bit) with kernel version 2.6.34

3.6 Overview of the Optimized Code

Figure 5 gives an example of the optimized code while optimizations applied to loops. As shown in the figure, stack switching instructions are inserted at the entry and exit of the function. After switching to the 64-bit mode, the prologue starts to preload the values of selected stack variables into registers and the non-promoted stack variables are accessed through the shadow stack. Similarly, instructions to restore values from registers to memory appear in the loop epilogue. After the epilogue, there are instructions to switch operating mode back to compatibility mode.

4. Evaluation Environment

In this section, we present the hardware configuration, the experimental framework, the benchmarks, the compiler and the optimization options used in generating binaries for our experiments.

4.1 Hardware Configuration

We evaluate our mechanism on a 4-way Intel Xeon server, and each processor is a quad-core 2.27GHz Intel Xeon E5520. The configurations of the system are shown in Table 1.

4.2 Experimental Framework

DRPIE (Dynamic Register Promotion for IA-32 Executables) is a dynamic optimization system which optimizes IA-32 applications on the Intel64 platform at user level. Figure 6 illustrates the frame-

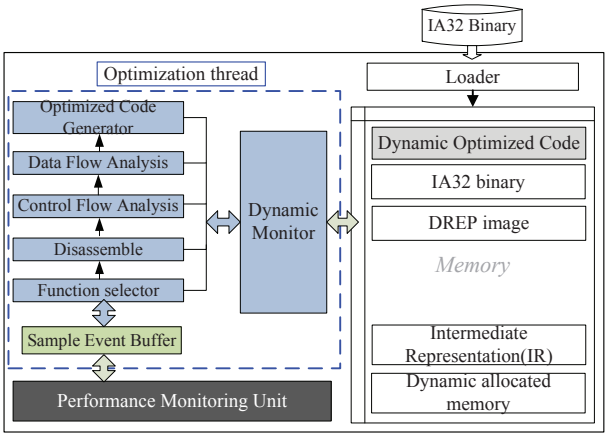


Figure 6. The framework of DRPIE

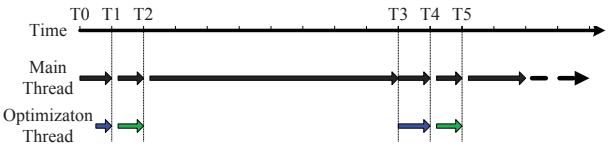


Figure 7. How the DRPIE thread works

work of DRPIE. There are two threads running in DRPIE, one is the original main thread running the IA-32 executable; the other is a dynamic optimization thread in charge of hotspot detection and runtime optimization. Figure 7 illustrates how the two threads work in DRPIE. As shown in the figure, the dynamic optimization thread first samples the main thread to locate the time consuming functions. At interval T1, the optimization thread suspends the main thread, instruments the legacy executable to obtain the execution frequency of the hot functions, and then resume the main thread. The main thread is continuously sampled during the interval T2. At the end of the interval T2, the dynamic optimizer selects some functions with a relatively large invocation time as optimization targets, and performs stack variable promotion. From T2 to T3, only the optimized main thread is running, no sampling and no instrumentation. At the end of interval T3, the dynamic optimization thread is activated again, and the performance sampling and instrumentation resume and the dynamic optimization process is repeated.

For each selected target function, we first construct a control-flow graph and identify loops in the function. Then we identify the stack variables in hot loops and select the favorable stack variables for register promotion. After that, we estimate the cost and benefit and determine whether we want to optimize for each loop or for the entire function. Finally, the optimized code is generated and patched to the original code.

4.3 Benchmarks

The SPEC CPU2006 suite is used as the benchmarks to measure the performance of DRPIE. All benchmarks were compiled by the Intel C++ Compiler 11.0 [8], and the *-fast* compiler optimization option is used. As discussed in Section 3.5, only the hot functions whose invocation time is greater than 0.001ms are optimized in our system. Therefore, some benchmarks may not have such functions for optimization. In the SPEC CPU2006 benchmark suite, 14 of them contain functions which are optimized, and we will present results for these benchmarks in the subsequent sections. The execution time we report in this section is an average of three identical runs.

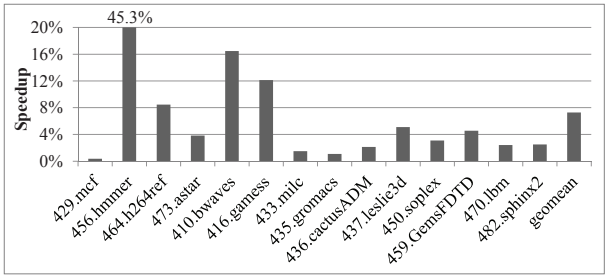


Figure 8. Performance improvement of dynamic register promotion

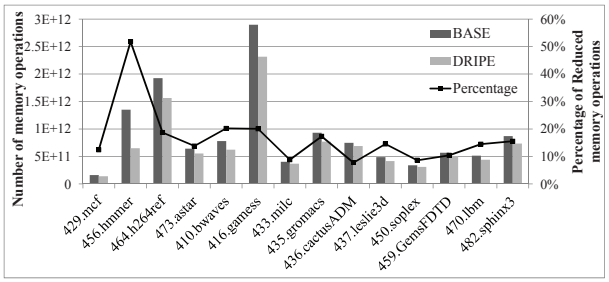


Figure 9. The number of memory operations before and after register promotion

5. Experimental results

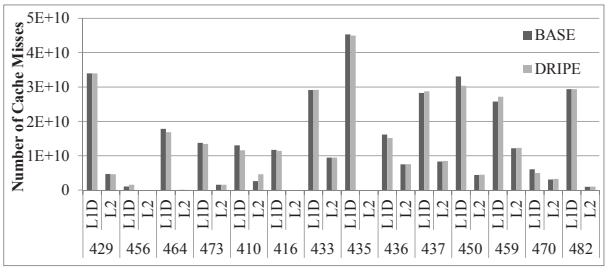
In this section, we present the performance evaluation of our DRPIE system. Through detailed analysis of the performance results observed by hardware performance counters, we verify that the observed performance gains are indeed attributed to the register promotion optimizations.

5.1 Overall performance improvement

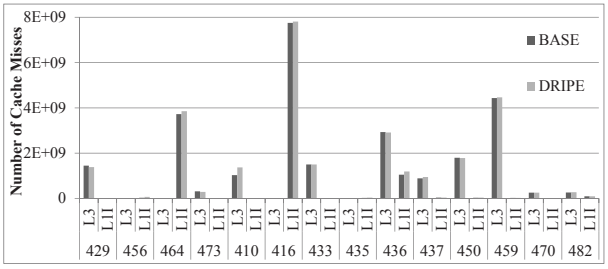
Figure 8 reports the performance improvement after promoting the stack variables in selected functions. The baseline performance is obtained by running the benchmarks on the same machine without our optimization. As the figure shows, some benchmarks have been substantially improved (45.3% for 456.hmmer, 16.4% for 410.bwaves and 12.1% for 416.gamess). On average, the performance gain is about 7.3%.

Figure 9 shows the number of memory operations before (i.e. BASE bars) and after register promotion (i.e. DRIPE bars), and the percentage of memory operation reduction. Compare to Figure 8, benchmarks with a large reduction of memory operations also get significant performance improvement. However, 429.mcf and 470.namd have more than 10% of memory operation reduction, but almost no performance gains. The reduction in memory references may not actually result in performance improvement because the load latency might have been hidden by dynamic instruction execution. If the promoted stack variables are not referenced on the critical path, the performance gain could be limited. Nevertheless, such programs may benefit from reduced power consumption (memory operations usually consume more power than simple ALU operations) [29].

Figure 10 shows the reduction in the number of cache misses. As we can see, there is no obvious decrease in the number of all-level cache misses for most benchmarks. There are eight GPRs and eight XMM registers, so we can promote at most eight 32-bit data and eight 128-bit data. This is about only 160 bytes of space, which accounts for a small fraction of the L1-D cache (about 0.5%).



(a) L1D and L2 cache misses



(b) L3 and L1I cache misses

Figure 10. The number of cache misses before and after register promotion

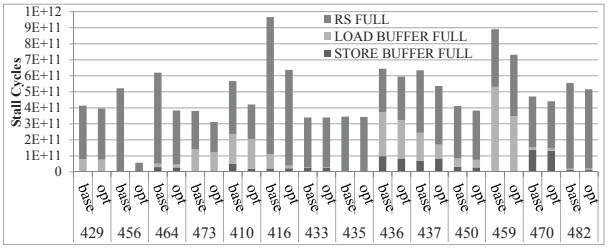


Figure 11. Stall cycles before and after register promotion

Therefore, the performance improvement is not due to the reduction in cache misses.

Figure 11 shows the number of stall cycles before and after register promotion. We can see that the benchmarks which we gain performance have many fewer stall cycles. The stall cycles is attributed to less load buffer full, store buffer full, and reservation station full. As shown in the figure, most of the reduced stall cycles come from the reduction in stall cycles of reservation station full and load buffer full. After register promotion, many load and store operations are eliminated, so the number of stall cycles due to load buffer full would reduce. In Intel Nehalem processor, the reservation station full usually caused by long latency operations in the pipeline, which are possibly load operations that miss the L2 cache or instructions dependent upon other instructions pending in the pipeline. However, since there are no obvious changes in the number of cache misses, the reduction of reservation station full cycles is likely coming from stalls in the execution unit.

Figure 12 shows the number of ITLB and DTLB misses. As we can observe from the figure, all benchmarks have very few ITLB misses, and there are no visible changes in the number of DTLB misses.

5.2 Evaluation of different optimization targets

In Section 2.3.3, we have discussed the pros and cons of applying our optimization to loop or function. Figure 13 shows the performance evaluation of optimizing different targets. As shown in the figure, optimization on function level gives good performance on 416.gamess and 470.lbm. However, for some benchmarks, such as 456.hmmer, the performance improvement comes from loop level optimization, not function level. This is because some function calls outside the loops caused access violations and the optimized code is discarded. On average, optimization at loop level is generally better than on function level in our current system.

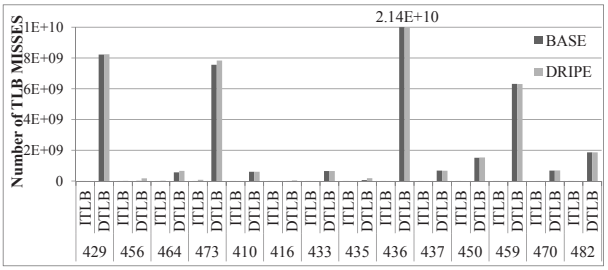


Figure 12. The number of TLB misses before and after register promotion

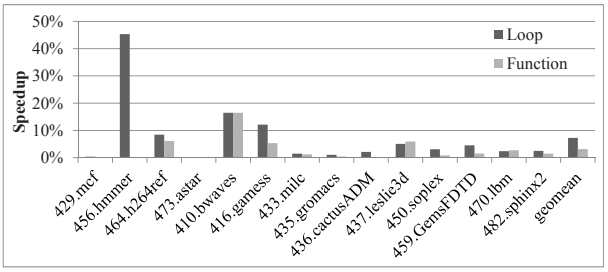


Figure 13. Performance for different optimization targets

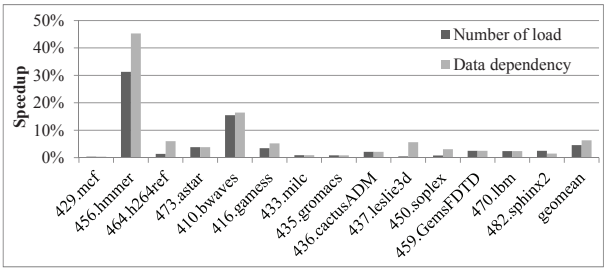


Figure 14. Performance with different selection criteria of stack variables

5.3 Evaluation of selection criteria of promotable variables

In Section 2.3.4, we have discussed two ways to select promotable variables with greater performance potential. Figure 14 shows the performance of the two selection criteria. We can see that, if we take critical path information of promotable variables into account, the performance would be significantly better for several benchmarks.

5.4 Performance impact of using SIMD registers as fast spill destination

In the newer Intel processors (such as Intel Nehalem), the cost of moving data between SIMD register and integer register is lower (about 1-2 cycles), so we can use the SIMD registers as fast spill destination. Figure 15 shows the performance impact of using

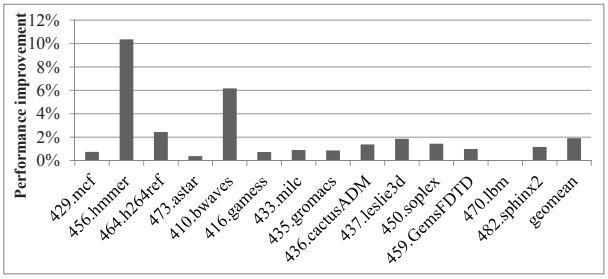


Figure 15. The performance impact of using SIMD registers as fast spill destination

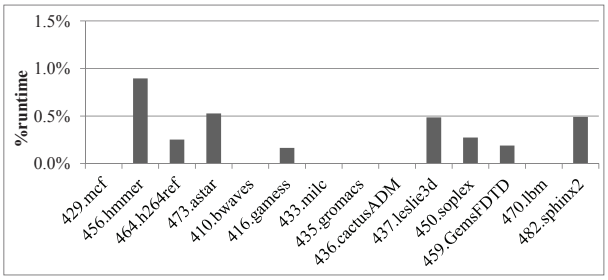


Figure 16. Run-time overhead of our DBT system and optimizations

SIMD registers for GPR spilling. As the figure shows, many benchmarks benefit from this optimization. On average, the performance improvement is about 1.9%.

5.5 Memory alias detection

For most benchmarks there are no implicitly memory accesses to the protected virtual pages. Among the 14 selected benchmarks, traps only occurred in 1 benchmarks (16 traps in 436.cactusADM) and they are all false memory aliases. Therefore, the overhead of the memory alias detection mechanism is low.

5.6 Analysis of run-time overhead

To get the runtime overhead of our system, we did everything but not really promote any stack variables into registers. All references to the stack variables are redirected to the shadow stack. Figure 16 shows the runtime overhead of our system. We can see that the runtime overhead is negligible for all benchmarks (only 0.9% at the most).

6. Related work

Register promotion allocates scalar variables to virtual registers in certain parts of a program to minimize the cost of load/store instructions. For example, Cooper and Lu [15] present a loop based register promotion algorithm. Sastry and Ju [22] propose a similar register promotion algorithm based on SSA form. Their algorithm inserts loads and stores in the enclosing interval, and then relying on the recursive promotion of outer intervals to propagate such loads and stores to a more appropriate interval. Sastry [23] proposes a method to use idle floating point resources (data path and registers) for integer programs. Their compiler identifies integer instructions that can be executed in floating-point unit and allocates integer variables in floating point registers. These algorithms are implemented in compilers and they do not promote a variable to register if prohibited by possible memory aliases. Our register promotion is performed at runtime, on legacy executables. Since it is difficult to disambiguate memory references at the binary level, we rely more on speculative register promotion rather than the traditional register promotion.

In the context of binary translation and optimization, Xu [28] aims to take advantage of the large number of registers in Itanium when they migrate IA-32 binaries to the Itanium processor. Their work assumes compiler annotations (called metadata section in their paper) on what variables can be promoted to registers are available. Our work is for legacy binaries where annotations or metadata are not available. Therefore, we adopt the approach of speculative register promotion, with a software based alias checking method using page protection and a novel stack switching mechanism.

Zhang et al [29] focus on profile-guided post-pass register promotion to utilize the dead/unused registers in hot regions of an object code. To deal with memory aliasing, they conduct a simple alias analysis to determine variables that are singly aliased (a conservative analysis which assumes all pointer variables are multiply aliased). Only load/store instructions corresponding to singly aliased, non-array variables are considered candidates for promotion. This method is safe, but too conservative and is likely to miss much optimization opportunities.

Postiff [21] proposes a new architecture support, the store-load address table, to speculatively promote variables to registers. Lin et al [13] actually use the ALAT (Advance Load Address Table) in the Itanium architecture to conduct speculative register promotion. Both works are performed in compilers. Our work is for legacy binaries, and it does not rely on hardware support. Our proposed method of alias checking for stack variables is software based, and can be applied to any architecture running Linux.

Wang et al [27] develop a dynamic binary translation system, called StarDBT, to translate IA-32 application binaries to 64-bit Intel64 code at run-time. Their experimental results show that the translated IA-32 applications suffer performance degradation due to high run-time overhead incurred by dynamic translation. Although our work also dynamically translates IA-32 binaries to Intel64 code, it translates only hot functions with register promotion opportunities.

In the context of dynamic binary optimization, there have been some prior work such as Dynamo [1], DynamoRIO [12] and Adore [16]. The optimizations applied in such systems include data cache prefetching, procedure inlining, partial dead code elimination, redundancy eliminations and code layout optimization. Since the above dynamic optimizers target at the same architecture, they do not have the opportunity to exploit additional registers as in our case.

7. Summary and conclusions

This paper focuses on utilizing the additional registers of the target architecture to improve the performance of source binaries in dynamic binary translation and optimization system. We have designed and developed a DBT system to promote stack variables in the source binary to the additional registers of the target architecture. One key challenge in this approach is how to promote stack variables to registers at the presence of possible memory aliasing. Our approach is to conduct register promotion speculatively by assuming there are no aliases between stack variables and pointer based references. We use a virtual page protection based mechanism in Linux and a novel stack switching method between explicit and implicit memory references to detect true aliases at runtime. Any detected true aliases would raise access violation traps, and our trap handler will undo the optimization for the offending function or loop.

We have implemented our proposed dynamic register promotion method on the Intel64 platform. Our DBT system has accel-

erated many benchmark programs in IA-32 code. Experimental results based on SPEC CPU2006 show that register promotion can be very effective for some benchmarks (e.g. 45.3% for 456.hmmer and 16.4% for 410.bwaves). We have conducted detailed performance analysis to understand the source of speed up. The main performance improvement comes from reduced stall on reservation stations and the load buffer. Although some of the benchmark programs do not benefit much from the optimizations, the observed runtime overhead of our system is extremely low (less than 1% in general) because our system selects only functions with substantial invocation time for optimization.

Previous work on speculative register promotion [13, 21] require architecture and hardware support. Our proposed approach is based on existing page protection mechanisms in virtual memory, and a novel stack switching method to minimize run-time traps. Our method can be applied to architectures with no special support for run-time alias checking. This paper has also provided sufficient engineering tradeoffs and details of our DBT system so that the ideas and experiments may be reproduced by others. Since memory aliasing constrains register promotion as well as instruction scheduling, this work could be extended to enable more speculative optimizations in DBT with our general runtime alias checking as a safety net.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, 2000. ISBN 1-58113-199-2.

[2] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. Ia-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium®-based systems. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 191, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X.

[3] D. Bruening, T. Garnett, and S. 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. ISBN 0-7695-1913-X.

[4] P. P. Bungale and C. Luk. Pinos: A programmable framework for whole-system dynamic instrumentation. In *Proceedings of the 3rd ACM/USENIX International Conference on Virtual Execution Environments*, pages 89–100, San Diego, June 2007. ACM.

[5] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates. Fx!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, 1998. ISSN 0272-1732.

[6] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developers Manual, Volume 1-3*. Intel, 2009. order number: 253665-030US.

[7] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*. Intel, 2009. order number: 248966-018.

[8] Intel Corporation. *Intel C++ Compiler User and Reference Guides*. Intel, 2009. order number: 307776-002US.

[9] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–24, 1998. ISBN 0-89791-979-3.

[10] D. Dhurjati and V. Adve. Efficiently detecting all dangling pointer uses in production servers. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*, pages 269–280, 2006. ISBN 0-7695-2607-1.

[11] S. Eranian. perfmon2: the hardware-based performance monitoring interface for Linux. `http://perfmon2.sourceforge.net/`, 2010. [Online; accessed 20-Sep-2010].

[12] T. Garnett. Dynamic optimization of ia-32 applications under dynamorio, 2003. M.S. thesis, Department of EECS, Massachusetts Institute of Technology (MIT).

[13] J. Lin, T. Chen, W.-C. Hsu, and P.-C. Yew. Speculative register promotion using advanced load address table (alat). In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 125–134, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X.

[14] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan. A compiler framework for speculative optimizations. *ACM Trans. Archit. Code Optim.*, 1(3):247–271, 2004. ISSN 1544-3566.

[15] J. Lu and K. D. Cooper. Register promotion in c programs. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 308–319, 1997. ISBN 0-89791-907-6.

[16] J. Lu, H. Chen, P.-C. Yew, and W. chung Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6:1–10, 2004.

[17] J. Lu, A. Das, W.-C. Hsu, K. Nguyen, and S. G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, 2005. ISBN 0-7695-2440-0.

[18] S. S. Muchnick. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Boston, MA, 1986.

[19] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, Elsevier, Singapore, 1997.

[20] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100, 2007. ISBN 978-1-59593-633-2.

[21] M. Postiff, D. Greene, and T. Mudge. The store-load address table and speculative register promotion. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, pages 235–244, 2000. ISBN 1-58113-196-8.

[22] A. V. S. Sastry and R. D. C. Ju. A new algorithm for scalar register promotion based on ssa form. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 15–25, 1998. ISBN 0-89791-987-4.

[23] S. S. Sastry, S. Palacharla, and J. E. Smith. Exploiting idle floating-point resources for integer execution. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 118–129, 1998. ISBN 0-89791-987-4.

[24] J. E. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, Elsevier, Singapore, 2005.

[25] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996. ISBN 0-89791-769-3.

[26] L. Torvalds. [PATCH] 2.6.1 (not 2.4.24!) mremap fixes broke shm alias mappings. `http://lkml.org/lkml/2004/1/12/265`, 2004. [Online; accessed 20-Sep-2010].

[27] C. Wang, S. Hu, H.-s. Kim, S. Nair, M. Breternitz, Z. Ying, and Y. Wu. Stardbt: An efficient multi-platform dynamic binary translation system. volume 4697, pages 4–15. Springer Berlin / Heidelberg, 2007.

[28] C. Xu, J. Li, T. Bao, Y. Wang, and B. Huang. Metadata driven memory optimizations in dynamic binary translator. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 148–157, 2007. ISBN 978-1-59593-630-1.

[29] K. Zhang, T. Zhang, and S. Pande. Binary translation to improve energy efficiency through post-pass register re-allocation. In *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*, pages 74–85, 2004. ISBN 1-58113-860-1.