Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation

Chi-Keung Luk Robert Cohn Robert Muth Harish Patil Artur Klauser Geoff Lowney
Steven Wallace Vijay Janapa Reddi † Kim Hazelwood

Intel Corporation †University of Colorado
Website: http://rogue.colorado.edu/Pin, Email: pin.project@intel.com

Abstract

Robust and powerful software instrumentation tools are essential for program analysis tasks such as profiling, performance evaluation, and bug detection. To meet this need, we have developed a new instrumentation system called *Pin*. Our goals are to provide *easy-to-use*, *portable*, *transparent*, and *efficient* instrumentation. Instrumentation tools (called *Pintools*) are written in C/C++ using Pin's rich API. Pin follows the model of ATOM, allowing the tool writer to analyze an application at the instruction level without the need for detailed knowledge of the underlying instruction set. The API is designed to be *architecture independent* whenever possible, making Pintools source compatible across different architectures. However, a Pintool can access architecture-specific details when necessary. Instrumentation with Pin is mostly *transparent* as the application and Pintool observe the application's original, uninstrumented behavior. Pin uses *dynamic compilation* to instrument executables while they are running. For efficiency, Pin uses several techniques, including inlining, register re-allocation, liveness analysis, and instruction scheduling to optimize instrumentation. This fully automated approach delivers significantly better instrumentation performance than similar tools. For example, Pin is 3.3x faster than Valgrind and 2x faster than DynamoRIO for basic-block counting. To illustrate Pin's versatility, we describe two Pintools in daily use to analyze production software. Pin is publicly available for Linux platforms on four architectures: IA32 (32-bit x86), EM64T (64-bit x86), Itanium®, and ARM. In the ten months since Pin 2 was released in July 2004, there have been over 3000 downloads from its website.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging-code inspections and walk-throughs, debugging aids, tracing; D.3.4 [*Programming Languages*]: Processors-compilers, incremental compilers

General Terms Languages, Performance, Experimentation

Keywords Instrumentation, program analysis tools, dynamic compilation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profi t or commercial advantage and that copies bear this notice and the full citation on the fi rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specifi c permission and/or a fee.

PLDI'05 June 12–15,2005,Chicago,Illinois,USA.
Copyright © 2005 ACM 1-59593-080-9/05/0006 . . . $5.00.

1. Introduction

As software complexity increases, *instrumentation*—a technique for inserting extra code into an application to observe its behavior—is becoming more important. Instrumentation can be performed at various stages: in the source code, at compile time, post link time, or at run time. *Pin* is a software system that performs run-time binary instrumentation of Linux applications.

The goal of Pin is to provide an instrumentation platform for building a wide variety of program analysis tools for multiple architectures. As a result, the design emphasizes *ease-of-use*, *portability*, *transparency*, *efficiency*, and *robustness*. This paper describes the design of Pin and shows how it provides these features.

Pin's instrumentation is *easy to use*. Its user model is similar to the popular ATOM [30] API, which allows a tool to insert calls to instrumentation at arbitrary locations in the executable. Users do not need to manually inline instructions or save and restore state. Pin provides a rich API that abstracts away the underlying instruction set idiosyncrasies, making it possible to write *portable* instrumentation tools. The Pin distribution includes many sample architecture-independent *Pintools* including profilers, cache simulators, trace analyzers, and memory bug checkers. The API also allows access to architecture-specific information.

Pin provides *efficient* instrumentation by using a just-in-time (JIT) compiler to insert and optimize code. In addition to some standard techniques for dynamic instrumentation systems including code caching and trace linking, Pin implements *register reallocation*, *inlining*, *liveness analysis*, and *instruction scheduling* to optimize jitted code. This fully automated approach distinguishes Pin from most other instrumentation tools which require the user's assistance to boost performance. For example, Valgrind [22] relies on the tool writer to insert special operations in their intermediate representation in order to perform inlining; similarly DynamoRIO [6] requires the tool writer to manually inline and save/restore application registers.

Another feature that makes Pin efficient is *process attaching and detaching*. Like a debugger, Pin can attach to a process, instrument it, collect profiles, and eventually detach. The application only incurs instrumentation overhead during the period that Pin is attached. The ability to attach and detach is a necessity for the instrumentation of large, long-running applications.

Pin's JIT-based instrumentation defers code discovery until run time, allowing Pin to be more *robust* than systems that use static instrumentation or code patching. Pin can seamlessly handle mixed code and data, variable-length instructions, statically unknown indirect jump targets, dynamically loaded libraries, and dynamically generated code.

Pin preserves the original application behavior by providing instrumentation *transparency*. The application observes the same ad-

dresses (both instruction and data) and same values (both register and memory) as it would in an uninstrumented execution. Transparency makes the information collected by instrumentation more relevant and is also necessary for correctness. For example, some applications unintentionally access data beyond the top of stack, so Pin and the instrumentation do not modify the application stack.

Pin's first generation, Pin 0, supports Itanium®. The recently-released second generation, Pin 2, extends the support to four[1] architectures: IA32 (32-bit x86) [14], EM64T (64-bit x86) [15], Itanium® [13], and ARM [16]. Pin 2 for Itanium® is still under development.

Pin has been gaining popularity both inside and outside of Intel, with more than 3000 downloads since Pin 2 was first released in July 2004. This paper presents an in-depth description of Pin, and is organized as follows. We first give an overview of Pin's instrumentation capability in Section 2. We follow by discussing design and implementation issues in Section 3. We then evaluate in Section 4 the performance of Pin's instrumentation and compare it against other tools. In Section 5, we discuss two sample Pintools used in practice. Finally, we relate Pin to other work in Section 6 and conclude in Section 7.

2. Instrumentation with Pin

The Pin API makes it possible to observe all the architectural state of a process, such as the contents of registers, memory, and control flow. It uses a model similar to ATOM [30], where the user adds procedures (as known as *analysis routines* in ATOM's notion) to the application process, and writes *instrumentation routines* to determine where to place calls to analysis routines. The arguments to analysis routines can be architectural state or constants. Pin also provides a limited ability to alter the program behavior by allowing an analysis routine to overwrite application registers and application memory.

Instrumentation is performed by a just-in-time (JIT) compiler. The input to this compiler is not bytecode, however, but a native executable. Pin intercepts the execution of the first instruction of the executable and generates ("compiles") new code for the straight-line code sequence starting at this instruction. It then transfers control to the generated sequence. The generated code sequence is almost identical to the original one, but Pin ensures that it regains control when a branch exits the sequence. After regaining control, Pin generates more code for the branch target and continues execution. Every time the JIT fetches some code, the Pintool has the opportunity to instrument it before it is translated for execution. The translated code and its instrumentation is saved in a code cache for future execution of the same sequence of instructions to improve performance.

In Figure 1, we list the code that a user would write to create a Pintool that prints a trace of address and size for every memory write in a program. The `main` procedure initializes Pin, registers the procedure called `Instruction`, and tells Pin to start execution of the program. The JIT calls `Instruction` when inserting new instructions into the code cache, passing it a handle to the decoded instruction. If the instruction writes memory, the Pintool inserts a call to `RecordMemWrite` *before* the instruction (specified by the argument `IPOINT_BEFORE` to `INS_InsertPredicatedCall`), passing the instruction pointer (specified by `IARG_INST_PTR`), effective address for the memory operation (specified by `IARG_MEMORYWRITE_EA`), and number of bytes written (specified by `IARG_MEMORYWRITE_SIZE`). Using

[1] Although EM64T is a 64-bit extension of IA32, we classify it as a separate architecture because of its many new features such as 64-bit addressing, a flat address space, twice the number of registers, and new software conventions [15].

```
FILE * trace;

// Print a memory write record
VOID RecordMemWrite(VOID * ip, VOID * addr, UINT32 size) {
    fprintf(trace,"%p: W %p %d\n", ip, addr, size);
}

// Called for every instruction
VOID Instruction(INS ins, VOID *v) {
    // instruments writes using a predicated call,
    // i.e. the call happens iff the store is
    // actually executed
    if (INS_IsMemoryWrite(ins))
        INS_InsertPredicatedCall(
            ins, IPOINT_BEFORE, AFUNPTR(RecordMemWrite),
            IARG_INST_PTR, IARG_MEMORYWRITE_EA,
            IARG_MEMORYWRITE_SIZE, IARG_END);
}

int main(int argc, char *argv[]) {
    PIN_Init(argc, argv);
    trace = fopen("atrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram(); // Never returns
    return 0;
}
```

Figure 1. A Pintool for tracing memory writes.

`INS_InsertPredicatedCall` ensures that `RecordMemWrite` is invoked only if the memory instruction is predicated `true`.

Note that the same source code works on all architectures. The user does not need to know about the bundling of instructions on Itanium, the various addressing modes on each architecture, the different forms of predication supported by Itanium and ARM, x86 string instructions that can write a variable-size memory area, or x86 instructions like `push` that can implicitly write memory.

Pin provides a comprehensive API for inspection and instrumentation. In this particular example, instrumentation is done one instruction at a time. It is also possible to inspect whole *traces*, *procedures*, and *images* when doing instrumentation. The Pin user manual [12] provides a complete description of the API.

Pin's *call-based* model is simpler than other tools where the user can insert instrumentation by adding and deleting statements in an intermediate language. However, it is equally powerful in its ability to observe architectural state and it frees the user from the need to understand the idiosyncrasies of an instruction set or learn an intermediate language. The inserted code may overwrite scratch registers or condition codes; Pin efficiently saves and restores state around calls so these side effects do not alter the original application behavior. The Pin model makes it possible to write efficient and architecture-independent instrumentation tools, regardless of whether the instruction set is RISC, CISC, or VLIW. A combination of inlining, register re-allocation, and other optimizations makes Pin's procedure call-based model as efficient as lower-level instrumentation models.

3. Design and Implementation

In this section, we begin with a system overview of Pin. We then discuss how Pin initially gains control of the application, followed by a detailed description of how Pin dynamically compiles the application. Finally, we discuss the organization of Pin source code.

3.1 System Overview

Figure 2 illustrates Pin's software architecture. At the highest level, Pin consists of a virtual machine (VM), a code cache, and an instru-

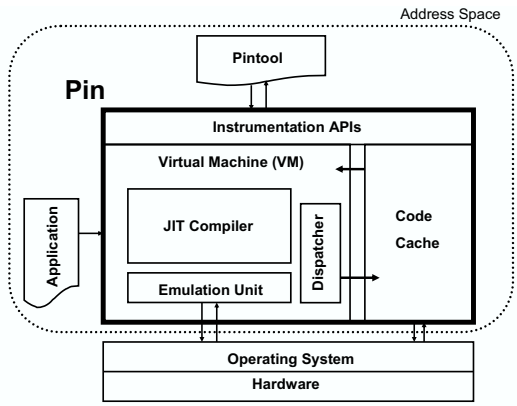

Figure 2. Pin's software architecture

mentation API invoked by Pintools. The VM consists of a just-in-time compiler (JIT), an emulator, and a dispatcher. After Pin gains control of the application, the VM coordinates its components to execute the application. The JIT compiles and instruments application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. Entering/leaving the VM from/to the code cache involves saving and restoring the application register state. The emulator interprets instructions that cannot be executed directly. It is used for system calls which require special handling from the VM. Since Pin sits above the operating system, it can only capture user-level code.

As Figure 2 shows, there are three binary programs present when an instrumented program is running: the application, Pin, and the Pintool. Pin is the engine that jits and instruments the application. The Pintool contains the instrumentation and analysis routines and is linked with a library that allows it to communicate with Pin. While they share the *same* address space, they do *not* share any libraries and so there are typically three copies of `glibc`. By making all of the libraries private, we avoid unwanted interaction between Pin, the Pintool, and the application. One example of a problematic interaction is when the application executes a `glibc` function that is not reentrant. If the application starts executing the function and then tries to execute some code that triggers further compilation, it will enter the JIT. If the JIT executes the same `glibc` function, it will enter the same procedure a second time while the application is still executing it, causing an error. Since we have separate copies of `glibc` for each component, Pin and the application do not share any data and cannot have a re-entrancy problem. The same problem can occur when we jit the analysis code in the Pintool that calls `glibc` (jitting the analysis routine allows us to greatly reduce the overhead of simple instrumentation on Itanium).

3.2 Injecting Pin

The injector loads Pin into the address space of an application. Injection uses the Unix Ptrace API to obtain control of an application and capture the processor context. It loads the Pin binary into the application address space and starts it running. After initializing itself, Pin loads the Pintool into the address space and starts it running. The Pintool initializes itself and then requests that Pin start the application. Pin creates the initial context and starts jitting the application at the entry point (or at the current PC in the case of attach). Using Ptrace as the mechanism for injection allows us to attach to an already running process in the same way as a debugger. It is also possible to detach from an instrumented process and continue executing the original, uninstrumented code.

Other tools like DynamoRIO [6] rely on the `LD_PRELOAD` environment variable to force the dynamic loader to load a shared library in the address space. First, `LD_PRELOAD` does not work with *statically-linked* binaries, which many of our users require. Second, loading an extra shared library will shift all of the application shared libraries and some dynamically allocated memory to a higher address when compared to an uninstrumented execution. We attempt to preserve the original behavior as much as possible. Third, the instrumentation tool cannot gain control of the application until after the shared-library loader has partially executed, while our method is able to instrument the very first instruction in the program. This capability actually exposed a bug in the Linux shared-library loader, resulting from a reference to uninitialized data on the stack.

3.3 The JIT Compiler

3.3.1 Basics

Pin compiles from one ISA directly into the same ISA (e.g., IA32 to IA32, ARM to ARM) without going through an intermediate format, and the compiled code is stored in a software-based code cache. Only code residing in the code cache is executed—the original code is never executed. An application is compiled one *trace* at a time. A trace is a straight-line sequence of instructions which terminates at one of the conditions: (i) an *unconditional* control transfer (branch, call, or return), (ii) a pre-defined number of *conditional* control transfers, or (iii) a pre-defined number of instructions have been fetched in the trace. In addition to the last exit, a trace may have multiple side-exits (the conditional control transfers). Each exit initially branches to a *stub*, which re-directs the control to the VM. The VM determines the target address (which is statically unknown for indirect control transfers), generates a new trace for the target if it has not been generated before, and resumes the execution at the target trace.

In the rest of this section, we discuss the following features of our JIT: trace linking, register re-reallocation, and instrumentation optimization. Our current performance effort is focusing on IA32, EM64T, and Itanium, which have all these features implemented. While the ARM version of Pin is fully functional, some of the optimizations are not yet implemented.

3.3.2 Trace Linking

To improve performance, Pin attempts to branch directly from a trace exit to the target trace, bypassing the stub and VM. We call this process *trace linking*. Linking a *direct* control transfer is straightforward as it has a unique target. We simply patch the branch at the end of one trace to jump to the target trace. However, an *indirect* control transfer (a jump, call, or return) has multiple possible targets and therefore needs some sort of target-prediction mechanism.

Figure 3(a) illustrates our indirect linking approach as implemented on the x86 architecture. Pin translates the indirect jump into a move and a direct jump. The move puts the indirect target address into register `%edx` (this register as well as the `%ecx` and `%esi` shown in Figure 3(a) are obtained via register re-allocation, as we will discuss in Section 3.3.3). The direct jump goes to the first predicted target address `0x40001000` (which is mapped to `0x70001000` in the code cache for this example). We compare `%edx` against `0x40001000` using the `lea`/`jecxz` idiom used in DynamoRIO [6], which avoids modifying the conditional flags register `eflags`. If the prediction is correct (i.e. `%ecx`=0), we will branch to `match1` to execute the remaining code of the predicted target. If the prediction is wrong, we will try another predicted target `0x40002000` (mapped to `0x70002000` in the code cache). If the target is not found on the chain, we will branch to `LookupHtab_1`, which searches for the target in a hash table (whose base address is

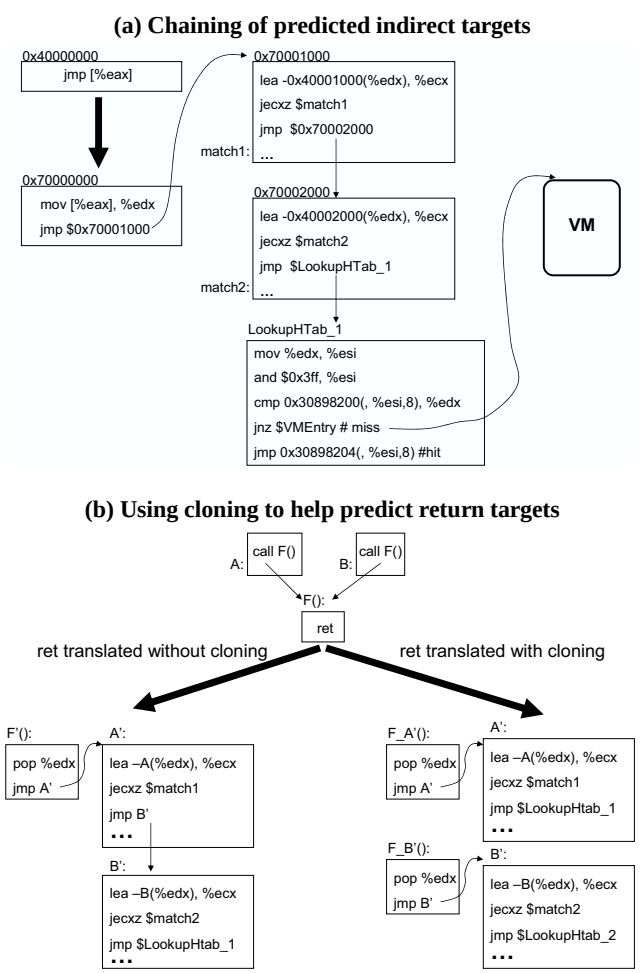

Figure 3. Compiling indirect jumps and returns

`0x30898200` in this example). If the search succeeds, we will jump to the translated address corresponding to the target. If the search fails, we will transfer to the VM for indirect target resolution.

While our indirect linking mechanism is similar to the approach taken in DynamoRIO [6], there are three important differences. First, in DynamoRIO, the entire chain is generated at one time and embedded at the translation of the indirect jump. Therefore no new predicted target can be added onto the chain after it is generated. In contrast, our approach *incrementally* builds the chain while the program is running and thus we can insert newly seen targets onto the chain *in any order* (e.g., Pin can put a new target either at the front or the end of the chain). These new targets can be found in the chain the next time that they occur, without searching the hash table. The second difference is that DynamoRIO uses a *global* hash table for all indirect jumps whereas Pin uses a *local* hash table for each individual indirect jump. A study by Kim and Smith [17] shows that the local hash table approach typically offers higher performance. The third difference is that we apply *function cloning* [10] to accelerate the most common form of indirect control transfers: *returns*. If a function is called from multiple sites, we clone multiple copies of the function, one for each call site. Consequently, a return in each clone will have only one predicted target on the chain in most cases, as illustrated by the example in Figure 3(b). To implement cloning, we associate a *call stack* with each trace (more precisely to the *static context* of each trace, which we will discuss in Section 3.3.3). Each call stack remembers the last four call sites and is compactly represented by hashing the call-site addresses into a single 64-bit integer.

3.3.3 Register Re-allocation

During jitting, we frequently need extra registers. For example, the code for resolving indirect branches in Figure 3(a) needs three free registers. When instrumentation inserts a call into an application, the JIT must ensure that the call does not overwrite any scratch registers that may be in use by the application. Rather than obtaining extra registers in an ad-hoc way, Pin re-allocates registers used in both the application and the Pintool, using linear-scan register allocation [24]. Pin's allocator is unique in that it does interprocedural allocation, but must compile one *trace* at a time while incrementally discovering the flow graph during execution. In contrast, static compilers can compile one *file* at a time and bytecode JITs [5, 8] can compile a whole *method* at one time. We describe two issues that our trace-based register re-allocation scheme must address: *register liveness analysis* and *reconciliation of register bindings*.

Register Liveness Analysis Precise liveness information of registers at trace exits makes register allocation more effective since dead registers can be reused by Pin without introducing spills. Without a complete flow graph, we must incrementally compute liveness. After a trace at address A is compiled, we record the liveness at the beginning of the trace in a hash table using address A as the key. If a trace exit has a statically-known target, we attempt to retrieve the liveness information from the hash table so we can compute more precise liveness for the current trace. This simple method introduces negligible space and time overhead, yet is effective in reducing register spills introduced by Pin's register allocation.

Reconciliation of Register Bindings Trace linking (see Section 3.3.2) tries to make traces branch directly to each other. When registers are reallocated, the JIT must ensure than the register binding at the trace exit of the source trace matches the bindings of the entrance of the destination trace. A straightforward method is to require a standard binding of registers between traces. For example Valgrind [22] requires that all virtual register values be flushed to memory at the end of a basic block. This approach is simple but inefficient. Figure 4(b) shows how Valgrind would re-allocate registers for the original code shown in Figure 4(a). Here, we assume that virtual `%ebx` is bound to physical `%esi` in Trace 1 but to physical `%edi` in Trace 2. Virtual `%eax` and `%ebx` are saved at Trace 1's exit because they have been modified in the trace, and they are reloaded before their uses in Trace 2. `EAX` and `EBX` are the memory locations allocated by the JIT for holding the current values of virtual `%eax` and `%ebx`, respectively.

In contrast, Pin keeps a virtual register in the same physical register across traces whenever possible. At a trace exit e, if the target t has *not* been compiled before, our JIT will compile a new trace for t using the virtual-to-physical register binding at e, say B_e. Therefore, e can branch directly to t. Figure 4(c) shows how Pin would re-allocate registers for the same original code, assuming that target t has not been compiled before. Nevertheless, if target t has been previously compiled with a register binding $B_t \neq B_e$, then our JIT will generate compensation code [19] to reconcile the register binding from B_e to B_t instead of compiling a new trace for B_e. Figure 4(d) shows how Pin would re-allocate registers for the same original code, this time assuming that the target t has been previously compiled with a different binding in the virtual `%ebx`. In practice, these bindings show differences in only one or two virtual registers, and are therefore more efficient than Valgrind's method.

A design choice we encountered was *where* to put the compensation code. It could be placed *before* the branch, which is exactly the situation shown in Figure 4(d) where the two `mov` instructions

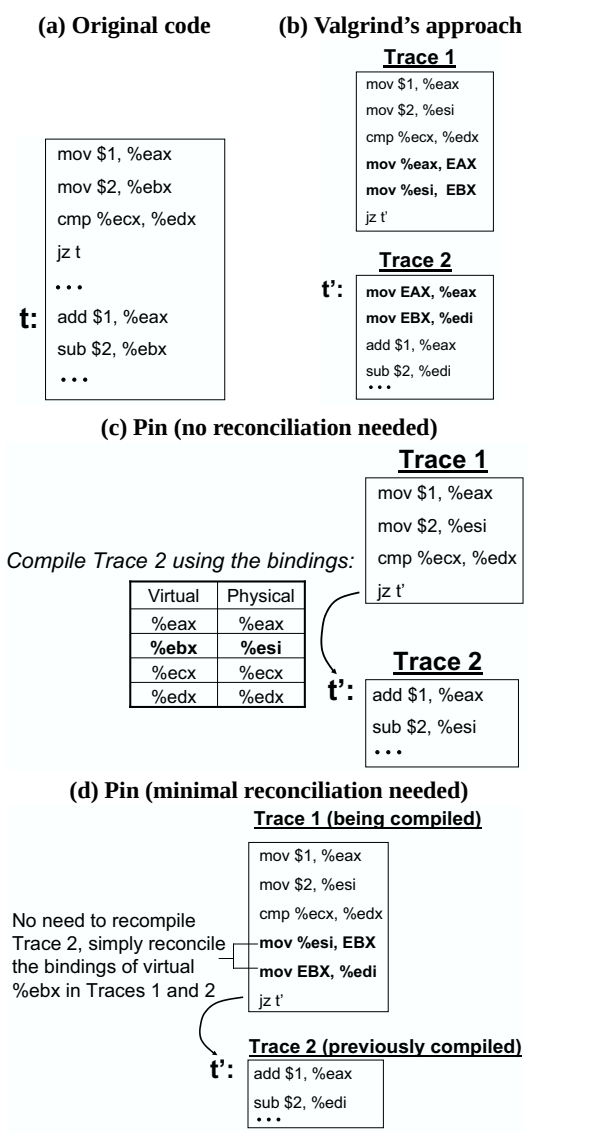

Figure 4. Reconciliation of Register Bindings

that adjust the binding are placed before the `jz`. Or the compensation code could be placed *after* the branch (in that case, the two `mov` instructions in Figure 4(d) would be placed in between the `jz` and t'). We chose the "before" approach because our experimental data showed that it generally resulted in fewer unique bindings, therefore reducing the memory consumed by the compiler. Placing the compensation code before the branch is equivalent to targeting the register allocation to match the binding at the branch target.

To support reconciliation of register bindings, we need to remember the binding at a trace's entry. This is done by associating each trace with a *static context (sct)*, which contains a group of static properties that hold at the trace's entry. Register binding is one such property; another example property is the call stack of the trace, which is used for function cloning (see Section 3.3.2). So, precisely speaking, a trace is defined as a pair $< entryIaddr, entrySct >$, where $entryIaddr$ is the original instruction address of the trace's entry and $entrySct$ is the static context of the trace. Before the JIT compiles a new trace, it will first search for a *compatible* trace in the code cache. Two traces are compatible if they have the same $entryIaddr$ and their $entrySct$'s are either identical or different in only their register bindings (in that case we can reconcile from one register binding to the other, as we have exemplified in Figure 4(d)). If a compatible trace is found, the JIT will simply use it instead of generating a new trace.

3.3.4 Thread-local Register Spilling

Pin reserves an area in memory for spilling virtual registers (e.g., `EAX` and `EBX` shown in Figure 4(b) are two locations in this spilling area). To support multithreading, this area has to be thread local. When Pin starts an application thread, it allocates the spilling area for this thread and steals a physical register (`%ebx` on x86, `%r7` on Itanium) to be the *spill pointer*, which points to the base of this area. From that point on, any access to the spilling area can be made through the spill pointer. When we switch threads, the spill pointer will be set to the spilling area of the new thread. In addition, we exploit an optimization opportunity coming from the *absolute addressing* mode available on the x86 architecture. Pin starts an application assuming that it is single threaded. Accesses to the spilling area are made through absolute addressing and therefore Pin does not need a physical register for the spill pointer. If Pin later discovers that the application is in fact multithreaded, it will invalidate the code cache and recompile the application using the spill pointer to access the spilling area (Pin can detect multithreading because it intercepts all thread-create system calls). Since single-threaded applications are more common than multithreaded ones, this hybrid approach works well in practice.

3.3.5 Optimizing Instrumentation Performance

As we will show in Section 4, most of the slowdown from instrumentation is caused by executing the instrumentation code, rather than the compilation time (which includes inserting the instrumentation code). Therefore, it is beneficial to spend more compilation time in optimizing calls to analysis routines. Of course, the runtime overhead of executing analysis routines highly depends on their invocation frequency and their complexity. If analysis routines are complex, there is not much optimization that our JIT can do. However, there are many Pintools whose frequently-executed analysis routines perform only simple tasks like counting and tracing. Our JIT optimizes those cases by *inlining* the analysis routines, which reduces execution overhead as follows. Without inlining, we call a *bridge routine* that saves all caller-saved registers, sets up analysis routine arguments, and finally calls the analysis routine. Each analysis routine requires two calls and two returns for each invocation. With inlining, we eliminate the bridge and thus save those two calls and returns. Also, we no longer explicitly save caller-saved registers. Instead, we rename the caller-saved registers in the inlined body of the analysis routine and allow the register allocator to manage the spilling. Furthermore, inlining enables other optimizations like constant folding of analysis routine arguments.

We perform an additional optimization for the x86 architecture. Most analysis routines modify the conditional flags register `eflags` (e.g., if an analysis routine increments a counter). Hence, we must preserve the original `eflags` value as seen by the application. However, accessing the `eflags` is fairly expensive because it must be done by pushing it onto the stack[2]. Moreover, we must switch to another stack before pushing/popping the `eflags` to avoid changing the application stack. Pin avoids saving/restoring `eflags` as much as possible by using *liveness analysis* on the `eflags`. The liveness analysis tracks the individual bits in the `eflags` written and read by each x86 instruction. We frequently discover that the

[2] On IA32, we can use `lahf`/`sahf` to access the `eflags` without involving the stack. However, we decided not to use them since these two instructions are not implemented on current EM64T processors.

Architecture	Number of Source Files	Number of Lines (including comments)
Generic	87 (48%)	53595 (47%)
x86 (IA32+EM64T)	34 (19%)	22794 (20%)
Itanium	34 (19%)	20474 (18%)
ARM	27 (14%)	17933 (15%)
TOTAL	182 (100%)	114796 (100%)

Table 1. Distribution of Pin source among different architectures running Linux. Over 99% of code is written in C++ and the remaining is in assembly.

	Hardware	Linux	Compiler	Binary
IA32	1.7GHz Xeon™, 256KB L2 cache, 2GB Memory	2.4.9	gcc 3.3.2, -O3 for SPECint (except in eon where we use icc)	Shared
			icc 8.0 for SPECfp	Static
EM64T	3.4GHz Xeon™, 1MB L2 cache, 4GB Memory	2.4.21	Intel ® compiler (icc 8.0), with interprocedural & profile-guided optimizations	Static
Itanium®	1.3GHz Itanium®2, 6MB L2 cache, 12GB Memory	2.4.18		Static
ARM	400 MHz XScale® 80200, 128 MB Memory	2.4.18	gcc 3.4.1, -O2	Static

Table 2. Experimental setup.

`eflags` are dead at the point where an analysis routine call is inserted, and are able to eliminate saving and restoring of the `eflags`.

Finally, to achieve even better performance, the Pintool writer can specify a hint (`IPOINT_ANYWHERE`) telling Pin that a call to an analysis routine can be inserted *anywhere* inside the scope of instrumentation (e.g., a basic block or a trace). Then Pin can exploit a number of optimization opportunities by *scheduling* the call. For instance, Pin can insert the call immediately before an instruction that overwrites a register (or `eflags`) and thereby the analysis routine can use that register (or `eflags`) without first spilling it.

3.4 Organization of Pin Source Code

Since Pin is a multi-platform system, source code sharing is a key to minimizing the development effort. Our first step was to share the basic data structures and intermediate representations with Ispike [20], a static binary optimizer we previously developed. Then we organized Pin source into generic, architecture dependent, or operating-system dependent modules. Some components like the code cache are purely generic, while other components like the register allocator contain both generic and architecture-dependent parts. Table 1 shows the distribution of Pin source among different architectures, in terms of number of source files and lines. We combine IA32 and EM64T in Table 1 since they are similar enough to share the same source files. The x86 numbers do not include the decoder/encoder while the Itanium numbers do not include the instruction scheduler. The reason is that we borrow these two components from other Intel tools in library form and we do not have their sources. The data reflects that we have done a reasonable job in code sharing among architectures as about 50% of code is generic.

4. Experimental Evaluation

In this section, we first report the performance of Pin without any instrumentation on the four supported architectures. We then report the performance of Pin with a standard instrumentation—basic-block counting. Finally, we compare the performance of Pin with two other tools: DynamoRIO and Valgrind, and show that Pin's instrumentation performance is superior across our benchmarks.

Our experimental setup is described in Table 2. For IA32, we use dynamically-linked SPECint binaries compiled with `gcc -O3`. We compiled `eon` with `icc` because the `gcc -O3` version does not work, even without applying Pin. We could not use the official statically-linked, `icc`-generated binaries for all programs because DynamoRIO cannot execute them. We ran the SPEC2000 suite [11] using reference inputs on IA32, EM64T, and Itanium. On ARM, we are only able to run the training inputs due to limited physical memory (128MB), even when executing uninstrumented binaries. Floating-point benchmarks are not used on ARM as it does not have floating-point hardware.

4.1 Pin Performance without Instrumentation

Figure 5 shows the performance of applying Pin to the benchmarks on the four architectures, *without* any instrumentation. Since Pin 2/Itanium is still under development, we instead use Pin 0 for Itanium experiments. The y-axis is the time normalized to the native run time (i.e. 100%). The slowdown of Pin 2 on IA32 and EM64T is similar. In both cases, the average run-time overhead is around 60% for integer and within 5% for floating point. The higher overhead on the integer side is due to many more indirect branches and returns. The slowdown of Pin 0 on Itanium follows the same trend but is generally larger than on IA32 and EM64T, especially for floating-point benchmarks. This is probably because Itanium is an in-order architecture, so its performance depends more on the quality of the jitted code. In contrast, IA32 and EM64T are out-of-order architectures that can tolerate the overhead introduced in the jitted code. Pin's performance on ARM is worse than the other three architectures because indirect linking (see Section 3.3.2) is not yet implemented and there are fewer computational resources (ILP and memory) available.

One downside of dynamic compilation is that the compilation time is directly reflected in the application's run time. To understand the performance impact of dynamic compilation, we divide the total run time into the components shown in Figures 5(a), (b), and (d) (Pin 0 source code is not instrumented and hence does not have the breakdown). *Code Cache* denotes the time executing the jitted code stored in the code cache. Ideally, we would like this component to approach 100%. We divide the JIT time into three categories: *JIT-Decode*, *JIT-Regalloc*, and *JIT-Other*. *JIT-Decode* is the time spent decoding and encoding instructions, which is a non-trivial task on the x86 architecture. *JIT-Regalloc* is the time spent in register re-allocation. *JIT-Other* denotes the remaining time spent in the JIT. The last component is *VM*, which includes all other time spent in the virtual machine, including instruction emulation and resolving mispredicted indirect control transfers.

As Figures 5 (a) and (b) show, the JIT and VM components on IA32 and EM64T are mostly small except in `gcc` and `perlbmk`. These two benchmarks have the largest instruction footprint in SPEC2000 and their execution times are relatively short. Consequently, there is insufficient code reuse for Pin to amortize its compilation cost. In particular, Pin pays a high cost in re-allocating registers compared to most other tools that do not re-allocate registers. Nevertheless, the advantages provided by register re-allocation outweigh its compilation overhead (e.g., register re-allocation makes it easy to provide Pin and Pintools more virtual registers than the number of physical registers supported by the hardware). In practice, the performance overhead is a serious concern only for long-running applications. In that case, we would have sufficient code reuse to amortize the cost of register re-allocation. Figure 5(d) shows a different trend for ARM, where the VM component is

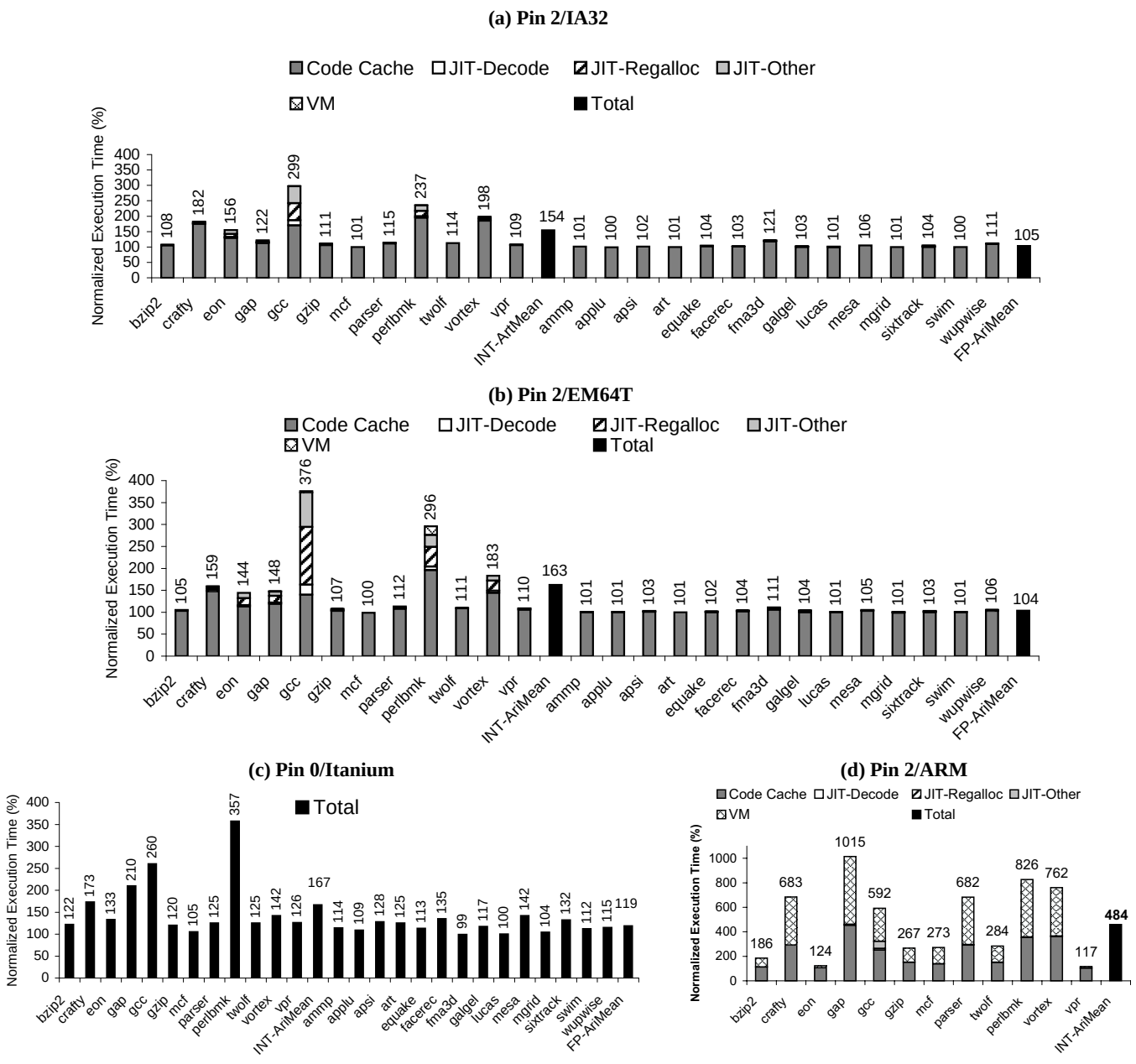

Figure 5. Performance of Pin (*without* any instrumentation) on four architectures. The y-axis is the time normalized to the native run time (i.e. 100%). INT-AriMean and FP-AriMean on the x-axis are the arithmetic means of the integer and floating-point benchmarks, respectively. The legends are explained in Section 4.1.

large but all JIT components are small. This is because register reallocation and indirect linking are not yet implemented on ARM. As a result, all indirect control transfers are resolved by the VM.

4.2 Pin Performance with Instrumentation

We now study the performance of Pin with basic-block counting, which outputs the execution count of every basic block in the application. We chose to measure this tool's performance because basic-block counting is commonly used and can be extended to many other tools such as `Opcodemix`, which we will discuss in Section 5.1. Also, this tool is simple enough that its performance largely depends on how well the JIT integrates it into the application. On the other hand, performance of a complex tool like detailed cache simulation mostly depends on the tool's algorithm. In that case, our JIT has less of an impact on performance.

Figure 6 shows the performance of basic-block counting using Pin on the IA32 architecture. Each benchmark is tested using four different optimization levels. Without any optimization, the overhead is fairly large (as much as 20x slowdown in `crafty`). Adding inlining helps significantly; the average slowdown improves from 10.4x to 7.8x for integer and from 3.9x to 3.5x for floating point. The biggest performance boost comes from the `eflags` liveness

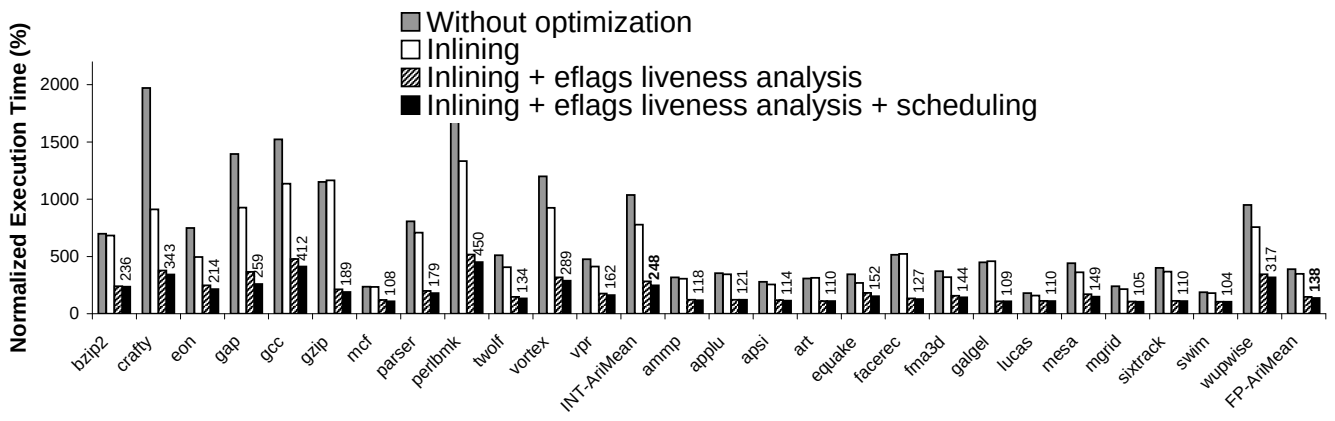

Figure 6. Performance of Pin with basic-block counting instrumentation on the IA32 architecture.

analysis, reducing the average slowdown to 2.8x for integer and 1.5x for floating point. Scheduling of instrumentation code further reduces the slowdown to 2.5x for integer and 1.4x for floating point.

4.3 Performance Comparison with Valgrind and DynamoRIO

We now compare the performance of Pin against Valgrind and DynamoRIO. Valgrind is a popular instrumentation tool on Linux and is the only binary-level JIT other than Pin that re-allocates registers. DynamoRIO is generally regarded as the performance leader in binary-level dynamic optimization. We used the latest release of each tool for this experiment: Valgrind 2.2.0 [22] and DynamoRIO 0.9.3 [6]. We ran two sets of experiments: one without instrumentation and one with basic-block counting instrumentation. We implemented basic-block counting by modifying a tool in the Valgrind package named `lackey` and a tool in the DynamoRIO package named `countcalls`. We show only the integer results in Figure 7 as integer codes are more problematic than floating-point codes in terms of the slowdown caused by instrumentation.

Figure 7(a) shows that without instrumentation both Pin and DynamoRIO significantly outperform Valgrind. DynamoRIO is faster than Pin on `gcc`, `perlbmk` and `vortex`, mainly because Pin spends more jitting time in these three benchmarks (refer back to Figure 5(a) for the breakdown) than DynamoRIO, which does not re-allocate registers. Pin is faster than DynamoRIO on a few benchmarks such as `crafty` and `gap` possibly because of the advantages that Pin has in indirect linking (i.e. incremental linking, cloning, and local hash tables). Overall, DynamoRIO is 12% faster than Pin without instrumentation. Given that DynamoRIO was primarily designed for *optimization*, the fact that Pin can come this close is quite acceptable.

When we consider the performance with instrumentation shown in Figure 7(b), Pin outperforms both DynamoRIO and Valgrind by a significant margin: on average, Valgrind slows the application down by 8.3 times, DynamoRIO by 5.1 times, and Pin by 2.5 times. Valgrind inserts a call before every basic block's entry but it does not automatically inline the call. For DynamoRIO, we use its low-level API to update the counter inline. Nevertheless, DynamoRIO still has to save and restore the `eflags` explicitly around each counter update. In contrast, Pin automatically inlines the call and performs liveness analysis to eliminate unnecessary `eflags` save/restore. This clearly demonstrates a main advantage of Pin: it provides efficient instrumentation without shifting the burden to the Pintool writer.

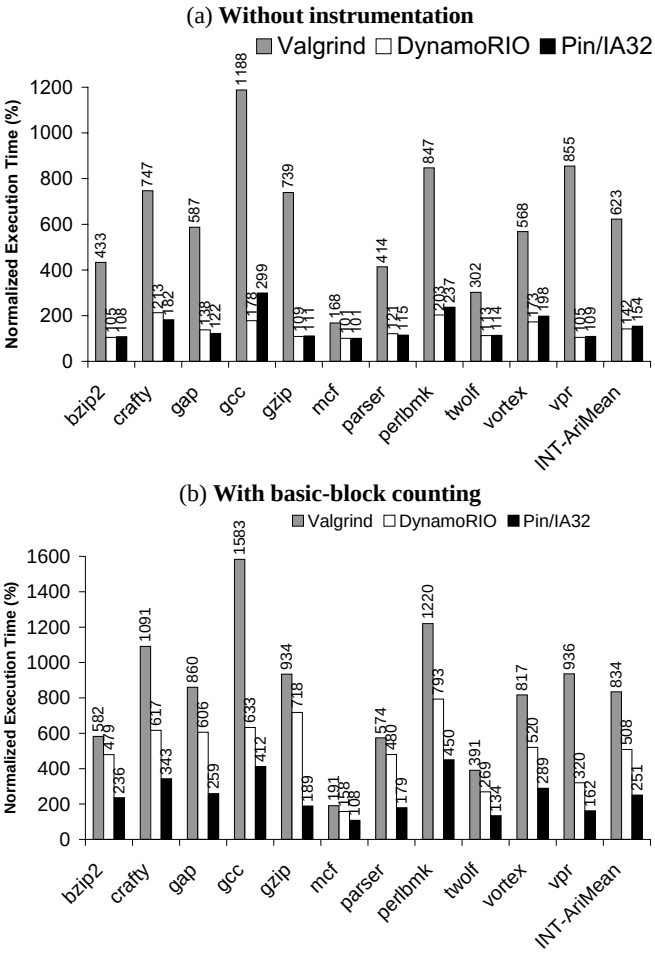

Figure 7. Performance comparison among Valgrind, DynamoRIO, and Pin. `Eon` is excluded because DynamoRIO does not work on the icc-generated binary of this benchmark. Omitting `eon` causes the two arithmetic means of Pin/IA32 slightly different than the ones shown in Figures 5(a) and 6.

5. Two Sample PinTools

To illustrate how Pin is used in practice, we discuss two Pintools that have been used by various groups inside Intel. The first tool, `Opcodemix`, studies the frequency of different instruction types in a program. It is used to compare codes generated by different compilers. The second tool, `PinPoints`, automatically selects representative points in the execution of a program and is used to accelerate processor simulation.

5.1 Opcodemix

`Opcodemix`, whose source code is included in the Pin 2 distribution [12], is a simple Pintool that can determine the dynamic mix of opcodes for a particular execution of a program. The statistics can be broken down on a per basic-block, per routine, or per image basis. Conceptually this tool is implemented as a basic-block profiler. We insert a counter at the beginning of each basic block in a trace. Upon program termination we walk through all the counters. From the associated basic-block's starting address, we can determine the function it belongs to and the instruction mix in that basic block. While the output of `Opcodemix` is ISA dependent (different ISAs have different opcodes), the implementation is generic—the same source code for `Opcodemix` is used on the four architectures.

Though simple, `Opcodemix` has been quite useful both for architectural and compiler comparison studies. As an example, the following analysis revealed a compiler performance problem. We collected `Opcodemix` statistics for the SPEC2000 images produced by two compilers, which we refer to as compilers A and B, for the EM64T architecture. For the benchmark `crafty`, we found that the image produced by compiler A executed 2% more dynamic instructions than the image produced by compiler B. To understand the cause of the extra instructions, we looked at the instruction distribution of frequently-executed routines. The data for the routine `PopCnt()` is shown in Table 3, where opcodes with significantly different frequencies in the two compilers are marked with "←". Examining the `PopCnt()` codes from the two compilers revealed that the deltas in `JE` and `JNZ` were due to different code-layout decisions, and the delta in `MOVL` was due to different register selections. The most surprising finding was the extra `PUSHQ` and `POPQ` generated by compiler A. Figure 8 shows the `PopCnt()` code generated by compiler A. After communicating with compiler A's writers, we learned that the `push` and `pop` are used for stack alignment but are in fact unnecessary in this case. As a result, this performance problem is now fixed in the latest version of compiler A.

In addition to SPEC, we use `Opcodemix` to analyze the Oracle database performance. Typically, more than 10 "Oracle" processes run on the system, but we want to ignore the database startup and only observe a single process performing a transaction. We first run Oracle natively (i.e. without Pin) to startup the database. Next we attach Pin to a single database server process and have it perform a transaction while collecting a profile. Pin's dynamic just-in-time instrumentation allows us to avoid instrumenting the entire 60 MB Oracle binary, and the attach feature allows us to avoid instrumenting the database startup and the other processes.

5.2 PinPoints

The purpose of the `PinPoints` [23] toolkit is to automate the otherwise tedious process of finding regions of programs to simulate, validating that the regions are representative, and generating traces for those regions. There are two major challenges in simulating large commercial programs. First, these programs have long run times, and detailed simulation of their entire execution is too time consuming to be practical. Second, these programs often have large resource requirements, operating system and device-driver dependencies, and elaborate license-checking mechanisms, making it difficult to execute them on simulators. We address the first challenge using SimPoint [28]—a methodology that uses phase analysis for finding representative regions for simulation. For the second challenge, we use Pin to collect SimPoint profiles (which we call *PinPoints*) and instruction traces, eliminating the need to execute the program on a simulator. The ease of running applications with Pintools is a key advantage of the `PinPoints` toolkit. `PinPoints` has been used to collect instruction traces for a wide variety of programs; Table 4 lists some of the Itanium applications (SPEC and commercial), including both single-threaded and multi-threaded applications. As the table shows, some of the commercial applications are an order of magnitude larger and longer-running than SPEC, and fully simulating them would take years. Simulating only the selected PinPoints reduces the simulation time from years to days. We also validate that the regions chosen represent whole-program behavior (e.g., the cycles-per-instruction predicted by PinPoints is typically within 10% of the actual value [23]). Because of its high prediction accuracy, fast simulation time, and ease-of-use, PinPoints is now used to predict performance of large applications on future Intel processors.

Instruction Type	Count		
	Compiler A	Compiler B	Delta
*total	712M	618M	-94M
XORL	94M	94M	0M
TESTQ	94M	94M	0M
RET	94M	94M	0M
PUSHQ	94M	0M	-94M <-
POPQ	94M	0M	-94M <-
JE	94M	0M	-94M <-
LEAQ	37M	37M	0M
JNZ	37M	131M	94M <-
ANDQ	37M	37M	0M
ADDL	37M	37M	0M
MOVL	0M	94M	94M <-

Table 3. Dynamic instruction distribution in `PopCnt()` of `crafty` benchmark.

```
42f538 <PopCnt>:
42f538: push %rsi # unnecessary
42f539: xor %eax,%eax
42f53b: test %rdi,%rdi
42f53e: je 42f54c
42f540: add $0x1,%eax
42f543: lea 0xffffffffffffffff(%rdi),%rdx
42f547: and %rdx,%rdi
42f54a: jne 42f540
42f54c: pop %rcx  # unnecessary
42f54d: retq
```

Figure 8. `PopCnt()` code generated by compiler A.

6. Related Work

There is a large body of related work in the areas of instrumentation and dynamic compilation. To limit our scope of discussion, we concentrate on *binary instrumentation* in this section. At the highest level, instrumentation consists of *static* and *dynamic* approaches.

Static binary instrumentation was pioneered by ATOM [30], followed by others such as EEL [18], Etch [25], and Morph [31]. Static instrumentation has many limitations compared to dynamic instrumentation. The most serious one is that it is possible to mix code and data in an executable and a static tool may not have enough information to distinguish the two. Dynamic tools can rely on execution to discover all the code at run time. Other difficult

Program	Description	Code Size (MB)	Dynamic Count (billions)
SPECINT 2000	SPEC CPU2000 integer suite [11]	1.9 (avg.)	521
SPECFP 2000	SPEC CPU2000 fbating -point suite [11]	2.4 (avg.)	724
SPECOMP 2001	SPEC benchmarks for evaluating *multithreaded* OpenMP applications [26]	8.4	4800
Amber	A suite of bio-molecular simulation from UCSF [1]	6.2	3994
Fluent	Computational Fluid Dynamics code from Fluent Inc [2]	19.6	25406
LsDyna	A general-purpose transient dynamic fi nite element analysis program from Livermore Software Technology [3]	61.9	4932
RenderMan	A photo-realistic rendering application from Pixar [4]	8.5	797

Table 4. Applications analyzed with PinPoints. Column 3 shows the code section size of the application binary and shared libraries reported by the `size` command. Column 4 lists the dynamic instruction count for the longest-running application input.

problems for static systems are indirect branches, shared libraries, and dynamically-generated code.

There are two approaches to dynamic instrumentation: *probe-based* and *jit-based*. The probe-based approach works by dynamically replacing instructions in the original program with trampolines that branch to the instrumentation code. Example probe-based systems include Dyninst [7], Vulcan [29], and DTrace [9]. The drawbacks of probe-based systems are that (i) instrumentation is *not* transparent because original instructions in memory are overwritten by trampolines, (ii) on architectures where instruction sizes vary (i.e. x86), we cannot replace an instruction by a trampoline that occupies more bytes than the instruction itself because it will overwrite the following instruction, and (iii) trampolines are implemented by one or more levels of branches, which can incur a significant performance overhead. These drawbacks make *fine-grained* instrumentation challenging on probe-based systems. In contrast, the jit-based approach is more suitable for fine-grained instrumentation as it works by dynamically compiling the binary and can insert instrumentation code (or calls to it) anywhere in the binary. Examples include Valgrind [22], Strata [27], DynamoRIO [6], Diota [21], and Pin itself. Among these systems, Pin is unique in the way that it supports high-level, easy-to-use instrumentation, which at the same time is portable across four architectures and is efficient due to optimizations applied by our JIT.

7. Conclusions

We have presented Pin, a system that provides easy-to-use, portable, transparent, efficient, and robust instrumentation. It supports the IA32, EM64T, Itanium®, and ARM architectures running Linux. We show that by abstracting away architecture-specific details, many Pintools can work across the four architectures with little porting effort. We also show that the Pin's high-level, call-based instrumentation API does not compromise performance. Automatic optimizations done by our JIT compiler make Pin's instrumentation even more efficient than other tools that use low-level APIs. We also demonstrate the versatility of Pin with two Pintools, `Opcodemix` and `PinPoints`. Future work includes developing novel Pintools, enriching and refining the instrumentation API as more tools are developed, and porting Pin to other operating systems. Pin is freely available at `http://rogue.colorado.edu/Pin`.

Acknowledgments

We thank Prof. Dan Connors for hosting the Pin website at University of Colorado. The Intel Bistro team provided the x86 decoder/encoder and suggested the instruction scheduling optimization. Ramesh Peri implemented part of the Pin 2/Itanium instrumentation.

References

[1] *AMBER home page*. http://amber.scripps.edu/.

[2] *Fluent home page*. http://www.fluent.com/.

[3] *LS-DYNA home page*. http://www.lstc.com/.

[4] *RenderMan home page*. http://RenderMan.pixar.com/.

[5] A.-R. Adl-Tabatabai, J. Bharadwaj, D.-Y. Chen, A. Ghuloum, V. Menon, B. Murphy, M. Serrano, and T. Shpeisman. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 7(1):19–31, Feb 2003.

[6] D. L. Bruening. *Effi cient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, M.I.T. (http://www.cag.lcs.mit.edu/dynamorio/), September 2004.

[7] B. R. Buck and J. Hollingsworth. An api for runtime code patching. *Journal of High Performance Computing Applications*, 14(4):317–329, 2000.

[8] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for java. In *ACM Java Grande Conference*, pages 129–141, June 1999.

[9] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, 2004.

[10] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2), April 1993.

[11] J. L. Henning. SPEC CPU2000: measuring cpu performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[12] Intel. *Pin User Manual*. http://rogue.colorado.edu/Pin.

[13] Intel. *Intel Itanium Architecture Software Developer's Manual Vols 1-4*, Oct. 2002.

[14] Intel. *IA-32 Intel Architecture Software Developer's Manual Vols 1-3*, 2003.

[15] Intel. *Intel Extended Memory 64 Technology Software Developer's Guide Vols 1-2*, 2004.

[16] Intel. *Intel PXA27x Processor Family Developer's Manual*, April 2004.

[17] H.-S. Kim and J. Smith. Hardware support for control transfers in code caches. In *Proceedings of the 36th Annual ACM/IEEE International Symposium on Microarchitecture*, Dec 2003.

[18] J. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN 95 Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[19] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[20] Chi-Keung Luk, Robert Muth, Harish Patil, Robert Cohn, and Geoff Lowney. Ispike: A Post-link Optimizer for the Intel Itanium Architecture. In *Proceedings of the 2nd Conference on Code Generation and Optimization*, pages 15–26, 2004.

[21] J. Maebe, M. Ronsse, and K. De Bosschere. Diota: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials held in conjunction with PACT'02*, 2002.

[22] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proceedings of the 3rd Workshop on Runtime Verifi cation*. http://valgrind.kde.org/, 2003.

[23] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. Pinpointing representative portions of large intel itanium progams

with dynamic instrumentation. In *Proceedings of the 37th Annual ACM/IEEE International Symposium on Microarchitecture*, Dec 2004.

[24] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions. on Programming Languages and Systems*, 21(5):895–913, Sept 1999.

[25] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. Levy, B. Bershad, and B. Chen. Instrumentation and optimization of win32/intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, August 1997.

[26] H. Saito, G. Gaertner, W. Jones, R. Eigenmann, H. Iwashita, R. Liberman, M. van Waveren, and B. Whitney. Large system performance of spec omp2001 benchmarks. In *Proceedings of the 2002 Workship on OpenMP: Experiences and Implementation*, 2002.

[27] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and retargetable software dynamic translation. In *Proceedings of the 1st Conference on Code Generation and Optimization*, pages 36–47, 2003.

[28] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[29] A. Srivastava, A. Edwards, and H. Vo. Vulcan: Binary transformation in a distributed environment. Technical Report MSR-TR-2001-50, Microsoft Research, April 2001.

[30] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[31] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. In *Proceedings of the 16th Symposium on Operating System Principles*, October 1997.