

MULTICORE
PROGRAMMINGANALYZING
PARALLEL
PROGRAMS
WITH PIN

Moshe Bach, Mark Charney, Robert Cohn, Elena Demikhovskiy, Tevi Devor,
Kim Hazelwood, Aamer Jaleel, Chi-Keung Luk, Gail Lyons, Harish Patil, and Ady Tal,
Intel Corp.

Software instrumentation provides the means to collect information on and efficiently analyze parallel programs. Using Pin, developers can build tools to detect and examine dynamic behavior including data races, memory system behavior, and parallelizable loops.

A decade ago, systems with multiple processors were expensive and relatively rare; only developers with highly specialized skills could successfully parallelize server and scientific applications to exploit the power of multiprocessor systems. In the past few years, multicore systems have become pervasive, and more programmers want to employ parallelism to wring the most performance out of their applications.

Exploiting multiple cores introduces new correctness and performance problems such as data races, deadlocks, load balancing, and false sharing. Old problems such as memory corruption become more difficult because parallel programs can be nondeterministic. Programmers need a deeper understanding of their software's dynamic behavior to successfully make the transition from single to multiple threads and processes.

Instrumentation is one tool for collecting the information needed to understand programs. Instrumentation-based tools typically insert extra code into a program to record events during execution.¹⁻⁴ The cost of executing the extra code can be as low as a few cycles, enabling fine-grained observation down to the instruction level.

Pin² (www.pintool.org) is a software system that performs runtime binary instrumentation of Linux and Microsoft Windows applications. Pin's aim is to provide an instrumentation platform for building a wide variety of program analysis tools, called *pintools*. By performing the instrumentation on the binary at runtime, Pin eliminates the need to modify or recompile the application's source and supports the instrumentation of programs that dynamically generate code.

INSTRUMENTATION

Pin provides a platform for building instrumentation tools. A *pintool* consists of instrumentation, analysis, and callback routines.¹ *Instrumentation routines* inspect the application's instructions and insert calls to analysis routines. *Analysis routines* are called when the program executes an instrumented instruction and often perform ancillary tasks. The program invokes *callbacks* when an event occurs, for example, when it is about to exit.

Figure 1 shows a simple *pintool* that prints the memory addresses of all data a program reads or writes. `Instruc-`

tion is an instrumentation routine that Pin calls the first time the program executes an instruction, so the routine can specify how it should be instrumented. If the instruction reads or writes memory, this example pintool inserts a call to `Address`—an analysis routine—and directs Pin to pass it the memory reference’s effective address. Immediately before a memory reference executes, the program calls `Address`, which prints the address to a file. The program invokes a callback routine, `Fini`, when it exits. Instrumentation and callback routines are registered in the pintool’s main function.

Figure 1 demonstrates only a small part of the Pin API. Whereas the example uses an instrumentation routine that can only see a single instruction at a time, Pin lets instrumentation routines see instruction blocks or whole binaries. The argument to `Address` is an effective address, but Pin provides much more, including register contents (for example, value of R9), the instruction pointer (IP or PC), procedure argument values, and constants. The only callback used in the example is for program end, but Pin also provides callbacks to notify a pintool about shared library loads, thread creation, system calls, Unix signals, and Microsoft Windows exceptions.

Although the instrumentation in this example is very simple, it is sufficient for a variety of useful tools. Instead of writing addresses to a file, a tool could feed the addresses to a software model of a cache and compute the cache miss rate for the application. By watching all the references to a specific memory location, it is possible to find an erroneous write through a pointer that overwrites a value with 1/100th the overhead of doing the same analysis in a debugger.

Pin uses a just-in-time (JIT) compiler to insert instrumentation into a running application. The JIT compiler recompiles and instruments small chunks of binary instructions immediately prior to executing them. Pin stores the modified instructions in a software code cache where they execute in lieu of the original application instructions. The code cache allows Pin to generate code regions once and reuse them for the remainder of program execution, amortizing compilation costs. Pin’s average base overhead is 30 percent, and user-inserted instrumentation adds to the time.

```
#include <stdio.h>
#include "pin.H"
FILE trace;
VOID Address(VOID * addr) { fprintf(trace,"%p\n", addr); }
VOID Instruction(INS ins, VOID *v) {
    if (INS_IsMemoryRead(ins)) {
        INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(Address),
            IARG_MEMORYREAD_EA, IARG_END);
    }
    if (INS_IsMemoryWrite(ins)) {
        INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(Address),
            IARG_MEMORYWRITE_EA, IARG_END);
    }
}
VOID Fini(INT32 code, VOID *v) { fclose(trace); }
int main(int argc, char *argv[]) {
    PIN_Init(argc, argv);
    trace = fopen("pinatrace.out", "w");
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_AddFiniFunction(Fini, 0);
    PIN_StartProgram();
    return 0;
}
```

Figure 1. Pintool for printing all program memory read and write addresses.

```
typedef void (*malloc_funptr_t)(size_t size);
malloc_funptr_t app_malloc;
VOID * malloc_wrap(size_t size) {
    void * ptr = app_malloc(size);
    printf("\nMalloc %d return %p\n", size, ptr);
    return ptr;
}
VOID Image(IMG img, VOID *v) {
    RTN mallocRtn = RTN_FindByName(img, "malloc");
    if (RTN_Valid(mallocRtn)) {
        app_malloc=
            (malloc_funptr_t)RTN_ReplaceProbed(mallocRtn,AFUNPTR(m
            alloc_wrap));
    }
}
```

Figure 2. Pintool’s fragment for wrapping `malloc`.

In Pin’s high-performance *probe mode* option, the base overhead is near zero. The probe mode has a limited set of callbacks available and restricts tools to interposing wrapper routines for global functions. Figure 2 shows a pintool’s fragment that wraps calls to `malloc` so it can print the argument and return values. `Image` is an instrumentation routine that the program invokes every time a binary or shared library loads. It searches the binary for a function called `malloc` and replaces it with a call to `malloc_wrap`. When the pro-

gram calls `malloc`, `malloc_wrap` is called instead, which calls the application `malloc`, then prints the argument and return value. To avoid infinite recursion, the call to `malloc` from `malloc_wrap` should not be redirected, so we instead call the function pointer returned by `RTN_ReplaceProbed`. The data collected from this tool could be used to find a program that incorrectly freed the same memory twice or track down some code that allocated too much memory.

In probe mode, the program binary is modified in memory. Pin overwrites the entry point of procedures with jumps (called probes) to dynamically generated instrumentation. This code can invoke analysis routines or a replacement routine. When the replacement routine needs to invoke the original function, it calls a copy of the entry point (without the probe) and continues executing the original program.

In probe mode, Pin overwrites the entry point of procedures with jumps to dynamically generated instrumentation.

Instrumenting parallel programs

Instrumenting a parallel program is not very different from instrumenting single-threaded programs. Pin provides callbacks when a new thread or new process is created. Analysis routines can be passed a thread ID so it is possible to attribute recorded data—for example, a memory reference address—to the thread that performed the operation.

Instrumenting a multithreaded program does require some special care by the tool writer. When a pintool instruments a parallel program, the application threads execute the calls to analysis functions. If the pintool in Figure 1 is invoked on a multithreaded program, then all the application threads can call the `Address` function simultaneously.

The pintool author is responsible for making the analysis functions thread-safe so they can be applied to a multithreaded program. Writing a thread-safe analysis routine is similar to writing a thread-safe routine in a multithreaded program. Authors use locks to synchronize references to shared data with other threads.

Pin also provides APIs for allocating and addressing thread-local storage. For example, the `Address` function in Figure 1 writes the program address to a file. The `trace` variable points to a `FILE` descriptor, which all threads share. It is not safe for multiple threads to write to `FILE` simultaneously. To enable this pintool to correctly instrument a multithreaded program, the `Address` function must either have a lock around the call to `fprintf` or create a

separate output file for each thread and retrieve the file descriptor from thread-local storage.

Performance considerations

Correcting a parallel program by adding locks is usually straightforward. However, a highly contended lock serializes execution and leads to poor CPU utilization. Because application threads execute analysis routines, a highly contended lock in an analysis routine will also serialize the application's execution. The serialization increases the pintool's overhead when compared to the application's uninstrumented execution and might alter the parallel program's behavior drastically. Pintool authors must employ standard parallel programming techniques to avoid excessive serialization. They should use thread-local storage to avoid the need to lock global storage. Instead of a single monolithic lock for a data structure, they should use fine-grained locks.

False sharing is another pitfall in naïve pintools, occurring when multiple threads access different parts of the same cache line and at least one of them is a write. To maintain memory coherency, the computer must copy the memory from one CPU's cache to another, even though data is not truly shared. False sharing is less costly when CPUs operate out of a shared cache, as is true for the four cores in the Intel Core i7 processor. Developers can eliminate false sharing by padding critical data structures to the size of a cache line or rearranging the structures' data layout.

Multithreaded versus multiprocess instrumentation

Pin allows instrumentation of parallel programs that use multiple threads and multiple cooperating processes. The new thread executes the same instrumented code as the other threads and accesses the same data. When a program spawns a new process or a process exits, Pin notifies the pintool. The pintool can choose to let the new process execute natively or under its control. The new process will have new code that the pintool must reinstrument. The processes do not share pintool data; however, a pintool can use OS-provided mechanisms for communication between the parallel program's instrumented processes.

EXAMPLE TOOLS

Developers can use various Pin-based tools to analyze parallel program performance and correctness.

Intel Parallel Inspector

The Intel Parallel Inspector (<http://software.intel.com/en-us/intel-parallel-inspector>) analyzes the multithreaded programs' execution to find memory and threading errors, such as memory leaks, references to uninitialized data, data races, and deadlocks. Intel Parallel Inspector uses Pin to instrument the running program and collect the information necessary to detect errors.

A *data race* occurs when two threads access the same data, at least one access is a write, and there is no synchronization (for example, locking) between accesses.⁵ Unsynchronized variable writes usually are a programming error and can cause nondeterministic behavior.

To detect data races, Parallel Inspector uses Pin to instrument all machine instructions in the program that reference memory and records the effective addresses (similar to Figure 1). It also instruments calls to thread synchronization APIs. By examining the effective addresses, Intel Parallel Inspector can detect when multiple threads access the same data. The synchronization API's instrumentation lets Intel Parallel Inspector determine if the memory accesses were synchronized. To help the programmer identify the cause of the data race, Intel Parallel Inspector shows the source lines and the call stacks leading to the problematic memory references.

Pin provides an API for mapping a machine instruction address to the corresponding source line and file. A debugger provides a call stack by unwinding stack frames and recovering the procedure call return addresses from the stack. Error-checking tools need to record the call stack for every memory reference because the tools might not determine until later whether that reference caused an error. Unwinding the call stack for every reference is expensive. Instead, tools typically keep a shadow call stack. A pintool instruments all call instructions, saving the stack pointer's current value and the called procedure on the shadow stack. Procedure return instructions are also instrumented, popping off enough shadow stack frames to resynchronize with the stack pointer's current value.

Intel Parallel Amplifier


The Intel Parallel Amplifier (<http://software.intel.com/en-us/intel-parallel-amplifier>) performs three types of analysis to help programmers improve program performance: hotspots, concurrency, and locks and waits. *Hotspots* attribute time to source lines and call stacks, identifying the parts of the programs that would benefit from tuning and parallelism. *Concurrency* measures the CPUs' utilization, giving whole program and per-function summaries. *Locks and waits* measures the time multithreaded programs spend waiting on locks, attributing time to synchronization objects and source lines. Identifying locks responsible for wait time and the associated source lines helps programmers improve a parallel program's CPU utilization.

Hotspot and concurrency analysis data comes from sampling. Intel Parallel Amplifier uses Pin to instrument the application to collect data for the locks and waits analysis. Capturing accurate timing data requires low overhead instrumentation. The locks and waits analysis uses Pin's probe mode to replace calls to synchronization APIs with wrapper functions, as Figure 2 demonstrates. The wrapper functions call the original synchronization function

and record the wait time, synchronization object, and call stack. Because the Intel Parallel Amplifier only instruments synchronization routines and not every call and return, it cannot maintain a shadow call stack. Instead, the instrumentation unwinds the stack every time it needs to capture a call stack.

Intel Trace Analyzer and Collector

The Intel Trace Analyzer and Collector provides information critical to understanding and optimizing cluster performance by quickly finding performance bottlenecks with Message Passing Interface (MPI) communication. The tool presents a timeline showing when MPI messages are sent and received, and programmers can use this information to improve the CPU utilization. The Intel Trace Analyzer and Collector uses Pin's probe mode to instrument calls to the MPI library, collecting time stamps, arguments, and other data. If a user requests call stack information, a JIT-mode tool instruments call and return instructions to maintain a shadow stack.



Computational bandwidth increases faster than memory bandwidth, especially for multicore systems.

CMP\$im

Memory system behavior is critical to parallel program performance. Computational bandwidth increases faster than memory bandwidth, especially for multicore systems. Programmers must utilize as much bandwidth as possible for programs to scale to many processors. Hardware-based monitors can report summary statistics such as memory references and cache misses; however, they are limited to the existing cache hierarchy and are not well suited for collecting more detailed information such as the degree of cache line sharing or the frequency of cache misses because of false sharing.

CMP\$im⁶ uses Pin to collect the memory addresses of multithreaded and multiprocessor programs, then uses a memory system's software model to analyze program behavior. It reports miss rates, cache line reuse and sharing, and coherence traffic, and its versatile memory system model configuration can predict future systems' application performance. While CMP\$im is not publicly available, the Pin distribution includes the source for a simple cache model, `dcache.cpp`.

PinPlay

Debugging and analyzing parallel programs is difficult because their execution is not deterministic. The threads'

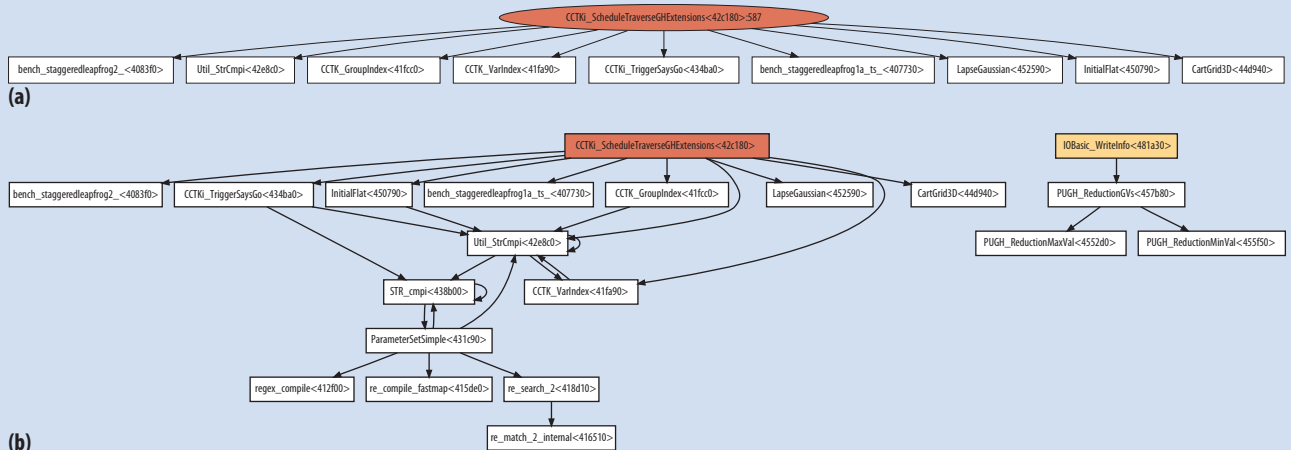


Figure 3. Visualization of Prospector's results: (a) call graph and (b) loop graph.

relative progress can change in every run of the program, possibly changing the results. Even single-threaded program execution is not deterministic because of behavior changes in certain system calls (for example, `gettimeofday()`) and stack and shared library load locations.

PinPlay is a Pin-based system for user-level capture and deterministic replay of multithreaded programs under Pin. The program first runs under the control of a Pin-based logging tool, which captures all the system call side effects⁷ and inter-thread shared-memory dependencies.⁸ Another Pin-based tool can replay the log, exactly reproducing the recorded execution by loading system call side effects and possibly delaying threads to satisfy recorded shared-memory dependencies.

Replaying a previously captured log by itself is not very useful. A pintool that instruments a program execution can also instrument a PinPlay log replay. The tool running off a PinPlay log sees the same program behavior on multiple runs, making the analysis deterministic. The program can also replay a PinPlay log while connected to a debugger, making multithreaded program debugging deterministic. As long as the PinPlay logger can capture a bug once, the behavior can repeat exactly multiple times with replay under a debugger. Future releases of Pin will include PinPlay.

Prospector

Compilers are ideal tools for exploiting parallelism because they can potentially perform automatic parallelization. However, even state-of-the-art compilers miss many parallelization opportunities in C/C++ programs, and as a result, programmers are forced to manually parallelize applications. The success of manual parallelization relies on execution profiler quality. Unfortunately, popular execution profilers, such as Gprof⁹ and Dev8 Partner (www.compuware.com), profile programs at function or

instruction granularity, which is insufficient for parallel programming because many programs are parallelized at the loop level.

To meet this need, developers created the Pin-based Prospector tool,¹⁰ which discovers potential parallelism in serial programs by *loop* and *data-dependence profiling*. Prospector provides loop execution profiles such as trip counts and the number of instructions executed inside loops. It also dynamically detects loop-carried data dependencies, which must be preserved during the parallelization process. Programmers receive reports on candidate loops for parallelization and can manually parallelize them with systems such as OpenMP (<http://openmp.org>) and Threading Building Blocks.¹¹ In addition to the profiler, Prospector provides several tools for visualizing and interpreting the profiling results.

Figure 3 shows the results of applying Prospector to the cactusADM program in the SPEC2006 benchmark suite. Figure 3a is the call graph displayed by Prospector. One of the functions, `CCTKi_ScheduleTraverseGHExtensions`, is highlighted because it contains a parallelizable loop. Figure 3b is this function's loop graph.

Intel Software Development Emulator

The Intel Software Development Emulator, or Intel SDE (www.intel.com/software/sde), is a user-level functional emulator for new instructions in the Intel64 instruction set built on Pin. Intel SDE supports emulation and debugging of multithreaded programs that use the Intel AVX (www.intel.com/software/avx), AES, and SSE4 instruction set extensions.

Whereas most tools use Pin to observe a program's execution, Intel SDE uses Pin to alter the program while it is running. During instrumentation, it deletes all instructions that must be emulated and replaces them with calls to functions that emulate the instruction.

Intel SDE is primarily a tool for debugging programs that use new instructions before the corresponding hardware exists. However, developers also can combine it with other tools to study performance. For example, combining Intel SDE and CMP\$im lets developers study the memory system behavior of programs that use new instructions.

PERFORMANCE AND SCALABILITY

Raw performance and scalability are both important properties when analyzing parallel applications. Pin's instrumentation performance is highly tied to the pinning application and even more so to the instrumentation routines added by the user. Applications running under the control of Pin scale as well as they do when running natively, and inserting heavy instrumentation code reduces this scalability.

Benchmarks and system details

For evaluation, we used a variety of workloads. Maxon Cinebench R10 is a photo-rendering workload that renders an image using a 3D scene file. POV-Ray is a publicly available ray-tracing package (www.povray.org). The SPEC benchmarks represent a suite of standardized workloads (www.spec.org). Finally, Illustrator and Excel are GUI applications that were exercised using Visual Test.

We ran our experiments on an Intel Xeon RW5580 (Intel Core i7) processor. The system has two sockets with quad-core processors and 6 Gbytes of memory and runs a 64-bit Windows 2003 Server with SP2. We disabled hyperthreading to simplify the scalability analysis, limiting the system to eight cores.

Runtime overhead

We first present the performance of a variety of tools that cover the spectrum of actual use of Pin from simple analysis to heavyweight tools with complex runtime analysis. Figure 4 shows the slowdown for various tools normalized to native execution.

The JIT configuration in Figure 4a executes the application under control of the JIT, with no instrumentation. It represents a lower bound for lightweight tools that need the observability of JIT-mode execution. BBCount introduces one counter increment per basic block. It is a lower bound for tools that only observe control flow. Its overhead increases as the average number of instructions per branch shrinks.

MemTrace records each memory address accessed by a thread in a thread-private buffer. This tool's source is in the Pin distribution. Cache simulators, memory corruption detectors, and data race detectors need to observe all memory addresses an application references, and MemTrace serves as a lower bound on their overhead. For these lightweight tools, Adobe Illustrator has

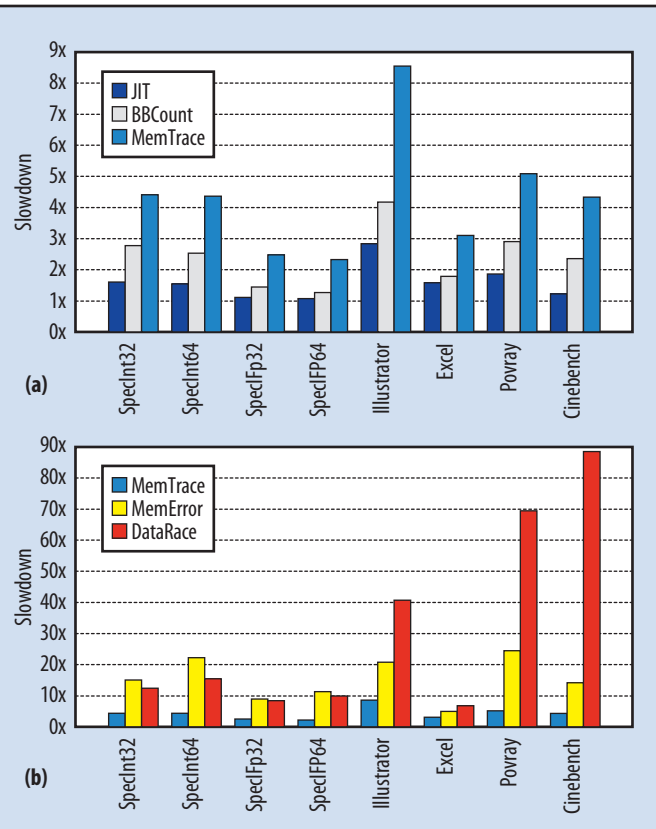


Figure 4. Slowdowns for a variety of tools from (a) lightweight analysis to (b) heavyweight analysis, normalized to native execution times.

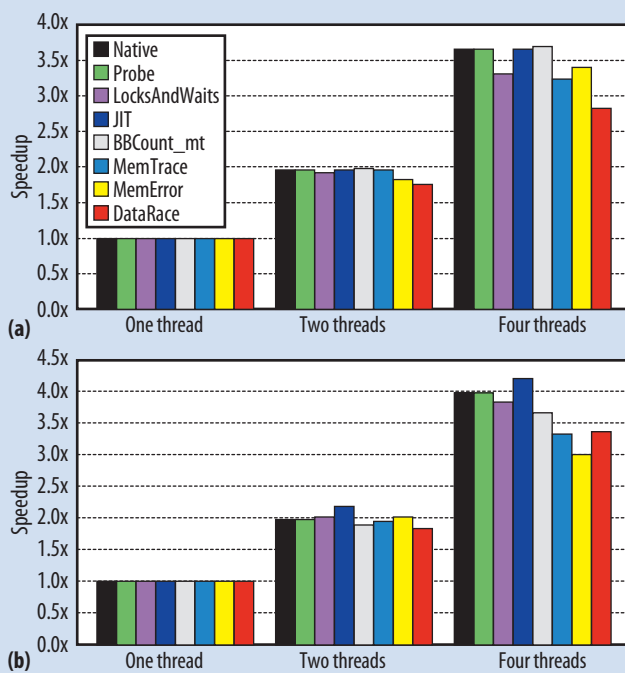
the highest overhead—it is a GUI application that has a large amount of low trip count code. Microsoft Excel is also a GUI application, but the workload allows a large amount of idle time, which hides overhead introduced by instrumentation.

Figure 4b shows slowdown for heavyweight tools. Like MemTrace, the MemError and DataRace tools primarily analyze memory references. DataRace records each memory address, but most of the overhead lies in the analysis—not address recording—explaining the higher overhead. MemError is the memory error analysis tool included in Intel Parallel Inspector. It checks for references to unallocated or uninitialized data. Whereas DataRace buffers addresses and checks them in batches, this tool immediately checks every address.

Figure 4b demonstrates that the slowdown for executing the program with Pin and no instrumentation (JIT) is small compared to the tools that do something useful (MemError and DataRace), and the time for the JIT configuration is not a good predictor for heavyweight tool time. Therefore, developers of commercial pintools generally focus on reducing the overhead of their own instrumentation routines, and the overhead of Pin itself is quite low by comparison.

Table 1. Maxon Cinebench rendering time in seconds for various pintools.

Pintool	1 thread	2 threads	4 threads
Native	197	100	54
Probe	197	100	54
LocksAndWaits	205	107	62
JIT	237	121	65
InsCount	470	241	128
MemTrace	851	436	263
DataRace	17,443	9,892	6,153
MemError	2,795	1,536	823

**Figure 5. Scalability of instrumented parallel applications (a) Cinebench and (b) POV-Ray.**

Scalable workload performance

We next explored Pin's performance scalability as it applies to additional threads and resources. Table 1 summarizes the time to render the Cinebench scene in seconds for various tools and numbers of threads. We ran the workload with one, two, and four rendering threads, setting the affinity to ensure that each thread was assigned its own CPU. For the two- and four-thread experiments, we set the affinity to place all the rendering threads on the same socket. This placement performs better than using multiple sockets because it lets all threads share the last-level cache. Cross-socket data sharing uses the memory bus, which is dramatically slower.

In Table 1, *Native* represents running the application without Pin. *Probe* represents executing the application

under the control of Pin in probe mode with no instrumentation. As expected, there is no overhead. *LocksAndWaits* is a probe-mode tool that instruments synchronization APIs and has relatively low overhead. *JIT* represents running the application in JIT mode with no instrumentation. The *InsCount* tool counts the number of instructions executed by adding instrumentation to every basic block to update the instruction count. Compared to JIT, *InsCount* overhead comes from the additional JIT time to generate instrumentation and maintain the counter. Finally, *MemTrace*, *DataRace*, and *MemError* represent high overhead tools as before. As Table 1 indicates, performance improves by nearly 4X with four threads.


Figure 5a shows the scaling in CPU time for a variety of configurations when running Cinebench. For each configuration, we normalize the time for two and four threads to the time for one thread for the same configuration. The native application scales well, achieving a 3.5x speedup on four processors. Figure 4b shows that complex analysis adds overhead, but the instrumented application is expected to scale with the addition of more threads. Running under the control of Pin with no instrumentation (JIT) achieves scalability similar to native. For the heavyweight tools, there is a slight drop-off in scalability. For *MemTrace*, limited memory bandwidth causes the reduced scalability. The other tools do more work per address, and memory bandwidth is less of a problem, but there is contention for tool-specific data structures. Figure 5b demonstrates similar trends in the POV-Ray benchmark. The native performance and JIT execution performance have almost perfect scaling. Instrumentation decreases the scaling.

Next, we explored the scalability of the *MemTrace* pintool when applied to the SPEC OMP2001 benchmarks. We varied the number of application threads from one to four, measured the time to execute under the control of Pin with no tool and with the *MemTrace* pintool, and normalized overhead to the native time.

JIT added minimal overhead and did not increase as the number of threads increased. This indicates that Pin does not introduce any serial overhead for these benchmarks; serial overhead would increase as more threads reduced the rest of the runtime. *MemTrace* added more overhead, more than doubling the running time. However, the overhead stayed the same or decreased as we added more threads, also showing that Pin preserves the program's parallelism.

In our final experiment, we compiled POV-Ray to use the new Intel AVX instructions, which extends SSE's vector capabilities. We used the Intel SDE tool to emulate the new instructions. The speedup for two and four processors was 2.1x and 3.4x when executed on an Intel Core i7 processor with four cores. We could not compare this to native execution because Intel does not yet sell processors that support the new instructions. The Intel

SDE tool did not introduce any shared data or synchronization, and, as expected, the scaling to four processors was excellent.

Pin was originally conceived as a tool for computer architecture analysis. A user community latched on to its flexible API and high performance, taking the tool into unforeseen domains like security, emulation, and parallel program analysis. We have learned that the interactions between multiple threads of control are difficult to analyze at compile time or to characterize with sampling techniques, making instrumentation's fine-grained, run-time analysis especially useful. Symbolic debuggers and hotspot profilers have been the primary tools for debugging and tuning sequential programs for the past 30 years. However, these tools are not sufficient for parallel programs. The flexibility of instrumentation with Pin enables new types of analysis such as data race or deadlock detectors. Pin puts the power in the hands of developers to craft the analysis that is appropriate for their domain or even application. 

Acknowledgments

The authors thank Douglas Armstrong, Zhiqiang Ma, Paul Petersen, and Ronen Zohar for their assistance in writing this article as well as the entire Pin team for making the article possible.

References

1. A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," *SIGPLAN Notices*, vol. 39, no. 4, ACM Press, 1994, pp. 528-539.
2. C.-K. Luk et al., "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, ACM Press, 2005, pp. 190-200.
3. V. Kiriansky, D. Bruening, and S.P. Amarasinghe, "Secure Execution via Program Shepherding," *Proc. 11th Usenix Security Symp.*, Usenix, 2002, pp. 191-206.
4. N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, ACM Press, 2007, pp. 89-100.
5. U. Banerjee et al., "A Theory of Data Race Detection," *Proc. Workshop Parallel and Distributed Systems: Testing and Debugging*, ACM Press, 2006, pp. 69-78.
6. A. Jaleel et al., "CMP\$im: A Pin-Based On-the-Fly Multi-core Cache Simulator," *Proc. 4th Ann. Workshop Modeling, Benchmarking and Simulation*, 2008, pp. 28-36.
7. S. Narayanasamy et al., "Automatic Logging of Operating System Effects to Guide Application-Level Architecture Simulation," *Proc. Joint Int'l Conf. Measurement and Modeling of Computer Systems*, ACM Press, 2006, pp. 216-227.
8. H. Patil et al., "PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs," *Proc. 6th Int'l Symp. Code Generation and Optimization*, ACM Press, 2010, pp. 1-10.
9. S.L. Graham, P.B. Kessler, and M.K. McKusick, "Gprof: A Call Graph Execution Profiler," *Proc. SIGPLAN 82 Symp. Compiler Construction*, ACM Press, 1982, pp. 120-126.
10. M. Kim, C.-K. Luk, and H. Kim, "Prospector: Discovering Parallelism via Dynamic Data-Dependence Profiling," tech. report TR-2009-001, Georgia Inst. of Technology, 2009.
11. J. Reinders, *Intel Threading Building Blocks*, O'Reilly, 2007.

Moshe (Maury) Bach leads the binary instrumentation team at Intel's Israel Design Center. He received a PhD in computing science from Columbia University. Contact him at mbach@iil.intel.com.

Mark Charney is a principal engineer at Intel. He received a PhD in electrical engineering from Cornell University. Contact him at mark.charney@intel.com.

Robert Cohn is a senior principal engineer at Intel. He received a PhD in computer science from Carnegie Mellon University. Contact him at robert.s.cohn@intel.com.

Elena Demikhovskiy is a senior software engineer at Intel's Israel Design Center. She received an MSc in computer science from Belarusian State University of Informatics and Radioelectronics. Contact her at elena.demikhovskiy@intel.com.

Tevi Devor is a senior software engineer at Intel. He received an MSc in computer science from Queens University, Kingston, Canada. Contact him at tevi.devor@intel.com.

Kim Hazelwood is an assistant professor at the University of Virginia and a faculty consultant for Intel. She received a PhD in computer science from Harvard University. Contact her at hazelwood@virginia.edu.

Aamer Jaleel is a hardware engineer at Intel. He received a PhD in electrical engineering from the University of Maryland. Contact him at aamer.jaleel@intel.com.

Chi-Keung Luk is a senior staff engineer at Intel. He received a PhD in computer science from the University of Toronto. Contact him at chi-keung.luk@intel.com.

Gail Lyons is a software engineer at Intel. She received an MSc in computer science from Boston University. Contact her at gail.lyons@intel.com.

Harish Patil is a senior staff engineer at Intel. He received a PhD in computer science from the University of Wisconsin, Madison. Contact him at harish.patil@intel.com.

Ady Tal is a software engineer and development team member from Intel. He received an MSc in computer science from The Technion—Israeli Institute of Technology. Contact him at ady.tal@intel.com.

 Selected CS articles and columns are available for free at <http://ComputingNow.computer.org>.