SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance

Steven Wallace
Intel Corporation

Kim Hazelwood
University of Virginia

Abstract

Dynamic instrumentation systems have proven to be extremely valuable for program introspection, architectural simulation, and bug detection. Yet a major drawback of modern instrumentation systems is that the instrumented applications often execute several orders of magnitude slower than native application performance. In this paper, we present a novel approach to dynamic instrumentation where several non-overlapping slices of an application are launched as separate instrumentation threads and executed in parallel in order to approach real-time performance. A direct implementation of our technique in the Pin dynamic instrumentation system results in dramatic speedups for various instrumentation tasks – often resulting in order-of-magnitude performance improvements. Our implementation is available as part of the Pin distribution, which has been downloaded over 10,000 times since its release.

1. Introduction

Understanding the characteristics of software applications is essential for effective system design, yet collecting details of an applications characteristics can be a tedious, error-prone process. Binary instrumentation tools allow users to easily, automatically, and correctly instrument existing compiled binaries. *Dynamic* instrumentation tools have the added ability to instrument not only the executing application, but all of the shared libraries that are invoked during the application's execution.

Dynamic instrumentation systems have been used for a variety of applications, from application introspection, to cache simulation drivers, to run-time security policy enforcement. In fact, one can envision countless reasons that a user may want to extend the functionality of an application at run time and/or in the absence of source code.

The single most significant drawback to dynamic instrumentation systems is their performance, which can vary from a 10% overhead for no instrumentation to the order of 1000X slowdown if the user wishes to embed very sophisticated and time-consuming functionality after every orig-

inal application instruction. The performance degradation can be attributed to two primary factors. The first factor is the overhead of the instrumentation system itself which can account for anywhere from a 10% to a 10X slowdown, depending on the code footprint, code reuse characteristics, application run time, and the efficiency of the instrumentation system. Reducing this first form of overhead has become well-charted territory in the research community [3, 5, 7, 9, 11, 12, 17, 22]. The second factor is the slowdown resulting from the extra functionality that the user wishes to insert into his/her application to collect data, perform simulations, etc., which can degrade performance well over a factor of 1000X. Until now, controlling the latter form of overhead has been considered the responsibility of the user. To our knowledge, our work is one of the first to directly tackle this second, more substantial form of overhead in a run-time system.

Our technique for reducing instrumentation overhead is to fundamentally change the way the dynamic instrumentation system handles instrumented code. Normally, dynamic instrumentation systems execute one copy of the instrumented application in a serial fashion (much in the same way that a normal application is executed). The alternative we explore is to execute an uninstrumented version of the code and routinely fork off instrumented *slices* of large code regions. These slices can then be executed in parallel with the main uninstrumented application process. Depending on the number of available processor cores and memory, and the size of each slice, we can approach the native, uninstrumented execution speed of the application.

As expected, our solution came with a large set of design and implementation decisions and challenges. Design decisions included the granularity and frequency at which we should spawn our instrumentation slices. Less frequent (longer) slices will minimize the forking overhead, but will increase the pipeline delay to complete the last slice. Furthermore, synchronizing multiple instrumentation slices, handling system calls, and merging results from the various instrumentation slices are but a few of the challenges we encountered during our implementation phase.

In this paper, we describe the overall design, implementation, performance, and applications of our parallelizing

version of the Pin dynamic instrumentation system, which we call SuperPin. Our implementation, including an extended API and sample applications, is currently available for download. The contributions of this paper include:

- The presentation of a novel approach to dynamic program instrumentation that has the potential to significantly improve instrumented code performance, to the point where it rivals native performance.
- A detailed description of the design and implementation challenges that surrounded a direct implementation of our approach.
- A full performance evaluation of our implementation.
- The presentation of our API that can be used to develop parallelizable instrumentation routines.
- A description of the potential applications and limitations of our system.

The remainder of the paper is organized as follows. Section 2 provides a detailed description of dynamic instrumentation systems, and Pin in particular. Section 3 then presents an overview of the SuperPin design. Section 4 discusses our experiences while implementing the SuperPin system, including the challenges we encountered, and our solutions to those challenges. Section 5 presents potential applications of SuperPin, presents our exported API, and provides sample uses of said API. Section 6 then includes a full performance evaluation of SuperPin and shows the significant speedups we gained from our direct implementation for longer-running applications. Finally, Section 7 summarizes our contributions and concludes, while Section 8 presents our plans for followup work.

2. Background

Before delving into the implementation details of SuperPin, we first provide an overview of dynamic instrumentation, its goals, and its applications. Then we provide a high-level view of the Intel Pin dynamic instrumentation system, upon which SuperPin was built.

2.1. Dynamic Instrumentation Systems

Dynamic instrumentation systems [17, 19] intercept the execution of an application in order to insert custom (user-defined) code at key points in the application, such as every instruction, branch, or function call, to name a few. This functionality has been very useful for workload analysis (counting instructions, branches, loads, etc.) but also for security [15] and reliability [21].

Internally, dynamic instrumentation systems are in many ways similar to dynamic optimization and dynamic translation systems. The primary goals of dynamic optimization

systems [3, 5, 7, 6] are to take advantage of additional information that was not available at compile time, such as program inputs or hardware configurations, and to adapt to run-time changes, such as stalls, phases, or temperature changes. Dynamic translation systems [2, 4, 8, 10, 13, 14, 16, 23] are designed to convert binaries for one instruction set into binaries for another instruction set on-the-fly. While the goals of each system are different, the core implementation is largely similar. In nearly all implementations, the system executes an *altered, cached copy* of the original application, and goes to great lengths to ensure that control is maintained at all times (despite interrupts, self-modifying code, indirect branches, etc.) and that the original, unmodified code is never executed.

2.2. Overview of Pin

Pin [17] is a dynamic binary instrumentation system developed at Intel. It consists of an instrumentation engine along with a large set of APIs that users can leverage to write their own custom plug-in utilities (called *Pintools*). Pin's API allows many plug-in Pintools to be source compatible across all supported instruction sets (IA-32, Intel64, IA-64, ARM) and operating systems (Linux, Windows, MacOS, FreeBSD). Pin's instrumentation engine allows a Pintool to insert function calls at any point in the program. It automatically saves and restores registers so the inserted call does not overwrite application registers. It also reallocates registers, inlines instrumentation, and caches previously modified code to improve performance.

Pin, the Pintool, and the application all execute in the same address space. The user invokes all three along with their corresponding run-time arguments on the command line: `pin -t pintool -- application`. Pin then uses `ptrace` to obtain control of the application and capture the processor context. Next, it loads and initializes the user's Pintool. Finally, Pin begins interpreting and/or compiling the application code, interleaving instrumentation as specified by the Pintool in a manner that is transparent to the executing application.

Pin's internal software architecture consists of a virtual machine (VM) and a code cache. The VM consists of a just-in-time compiler (JIT), an emulator, and a dispatcher, all of which work together to execute the instrumented application. The emulator interprets instructions that cannot be executed directly, such as system calls that require special handling from the VM. The dispatcher determines whether the next code region must be generated by the just-in-time compiler or is already present in the code cache.

A thorough description of the internal functionality of Pin is outside the scope of this paper, but is described by Luk et al. [17].
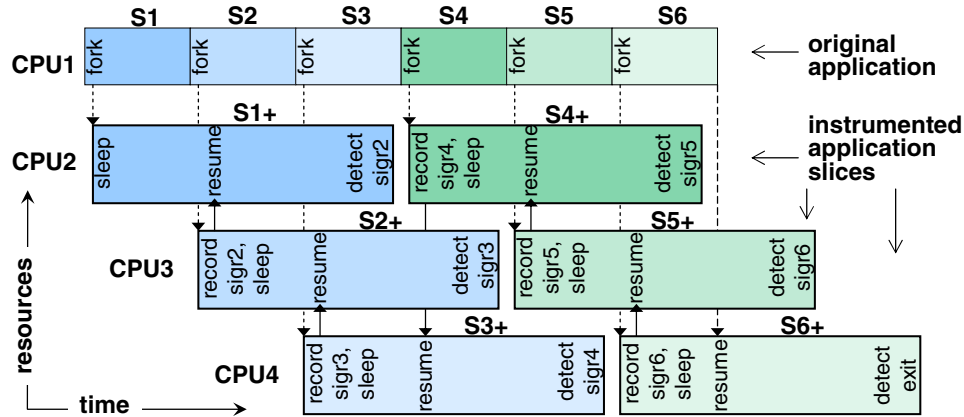
Figure 1. Our approach to dynamic instrumentation. SuperPin forks numerous instrumentation slices that execute in parallel on idle processor cores. Each slice sleeps until the following slice records its unique signature.

3. Parallelizing Instrumentation

SuperPin achieves significant speedup through parallelism. It achieves this parallelism by running the original application at full speed, while simultaneously executing distinct timeslices of that program under instrumentation[1]. After each timeslice ends, its results are merged to a collective total. The final result is displayed after the entire program has executed, and all instrumented timeslices have completed.

Figure 1 shows how the workload and timeslices are distributed across a four processor system. The illustration assumes the instrumented workload executes three times slower than the original application. CPU1 executes the original application. At the start of execution, the application forks off its first instrumented timeslice, depicted S1+. After some time has elapsed, the application will fork a second, non-overlapping instrumented timeslice, S2+.

In order to properly detect when the instrumented timeslice should end, S1+ will sleep until the second timeslice, S2+ is forked by the application. At that point, S2+ records a unique *signature*, which is a trigger for terminating the previous timeslice. S1+ will then resume execution until it reaches the S2 signature, i.e. the point where S2+ began. Meanwhile, the S2+ slice will sleep until S3+ records its signature, or an exit condition occurs.

Though the forked timeslices (S1+, S2+, ..., SN+) have a much longer runtime latency than their uninstrumented counterparts (S1, S2, ..., SN), they will all execute in a parallel manner as we show in Figure 1. This allows the

original application to progress at near real-time performance, while the longer instrumented timeslices will complete shortly thereafter.

After the last slice has been forked and the application has ended, there is a *pipeline delay* for the instrumentation slices to complete. If the system is fully loaded, this will take an extra $N * s$ seconds to finish, where N is the number of processors and s is the timeslice interval. If it is not fully loaded, it will take an extra $(F+1)s$ seconds, where F is the maximum number of simultaneous slices. For example, if $F = 7$ and $s = 1$ second, it will take 8 extra seconds for the instrumentation to finish. Hence, application times much greater than this are needed to show a significant improvement. We could reduce the timeslice size, but this can increase the overhead, as will be shown later. Nevertheless, as N is increased to greater than $F + 1$, the timeslice size is reduced to zero, and the application-induced overhead is reduced to zero, a close to real-time performance can be achieved.

Figure 1 shows fairly balanced timeslices for illustrative purposes. In reality, timeslice may vary in size, often significantly. Our approach is designed to handle this situation efficiently, and the signature detection mechanism becomes a crucial design element. We will clarify this point in Section 4.4.

4. Implementation Challenges

While the notion of parallelizing instrumentation is straightforward in theory, several hurdles were encountered during the implementation of SuperPin. In this section, we present some of the design decisions and challenges we encountered and describe our implemented solutions.

---

[1]While similar approaches have been used to instrument source code [20], our approach operates at the binary level, and is therefore automatic and transparent.

4.1. Spawning Instrumentation Slices

The key to SuperPin's performance improvement is to partition the application into instrumentable slices. Deciding *how* and *when* to create a slice thus falls into the realm of design decisions. We begin by describing *how* we spawn each instrumented timeslice.

SuperPin launches the application in the same manner as Pin. Pin performs an `exec` of the application and injects itself into the application space. As a result, the application and the instrumented slices all have their own copy of Pin's VM. They can communicate with each other via shared memory. Before each new timeslice is created, this shared memory space is created for the current and previous timeslices to share signature information.

When the control process determines that a new timeslice would be beneficial, it modifies the program counter to jump to a special trampoline. This trampoline changes the stack pointer to a private stack, then branches into the Pin VM, passing along information about the original program counter and stack.

As each slice is spawned, SuperPin will require memory for allocating the code cache and supportive data structures. These allocations should not interfere with the application's memory allocations, however, as that would violate transparency. To handle this situation, SuperPin allocates a large *bubble* of anonymous memory at the start of execution, which is used as a placeholder for the code cache structures. Then, immediately after spawning each slice, that memory is deallocated. Thus, any subsequent code cache allocations will occur in the bubble memory, away from the memory allocated by the application. This preserves *precise* memory mappings between the master and slices.

The next design decision we faced was determining *when* timeslices should be spawned. We found two natural boundaries for timeslice transitions: system calls and timeouts. Our final design includes both boundaries, therefore we describe each situation in the upcoming sections.

4.2. System Calls

When system calls are encountered in the master application, special care must be taken to ensure the children slices will behave in an identical manner. As a result, SuperPin employs a special control process that monitors the application via the `ptrace` mechanism.

After each system call, SuperPin must either (a) force a new slice or (b) record the effects of the system call and play them back in the slices. On some system calls, we perform custom emulation actions. For example, the `brk` system call can be duplicated without any adverse side effects. Meanwhile the anonymous `mmap` call can be repeated given the same address. In other cases where we are unsure

about the effects of a system call or encounter a new system call, SuperPin will default to forking a new timeslice.

For applications that have moderate system call usage, forking a new timeslice is acceptable. However, applications such as `gcc` will allocate and deallocate memory far too frequently. As a result, the overhead induced by forking becomes unacceptable. For these instances, we have implemented a record-and-playback mechanism. The memory modifications and results of system calls are recorded. The slices then playback the system call by changing the registers and modifying memory in an identical manner.

4.3. Timeouts

SuperPin also achieves timeslicing using a special timer process. If the control process has not forked a new slice due to a system call in a specified time interval (e.g, 1 second), it sends a stop signal to the master application. The control process then returns from its ptrace call and notices the timeout. It then creates a new slice using the trampolining technique described earlier.

4.4. Signature Detection

Each instrumented timeslice is responsible for detecting the correct point at which to end, such that it doesn't overlap with the subsequent timeslice. As mentioned earlier, this is accomplished via a unique *signature* that is recorded at the start of each timeslice that describes the system state. A key feature of SuperPin is its signature detection mechanism.

Timeslices that end as a result of a system call are straightforward to detect. However, some timeslices end as a result of a timeout at an arbitrary location. As a result, SuperPin needed a reliable mechanism that would uniquely identify a timeslice boundary. Furthermore, low overhead signature recording and detection mechanisms were essential. After examining several options, such as recording the last 1000 instruction pointers, we developed a relatively simple, yet effective signature-detection mechanism.

When a new slice has been created by forking the application, it begins instrumentation in *recording* mode. It records the state of the processor at a specific location. This includes the architectural register state and the top 100 words on the stack. The previous slice will then begin normal instrumentation and enter *detection* mode. Detection is only attempted (via instrumentation) at the specified instruction pointer. The detection routine first verifies a matching architectural state and then verifies a matching top-of-stack state.

To further optimize the detection process, the recorder attempts to ascertain the two registers that are most likely to change. For example, some registers are highly likely

to change over loop iterations. If the recorder cannot ascertain a clear candidate within a specified block count, then default registers are used. These two registers are checked first using the `INS_InsertIfCall` instrumentation. This will inline a quick check at that specific location. Only if that succeeds will the more expensive `INS_InsertThenCall` routine be called that performs a complete check.

We collected statistics on how frequently the quick and full detection is triggered. Only about 2% of the time does the quick detector trigger a full architectural state check. A stack check is usually only called once and succeeds. Very rarely is a stack check triggered more than once. This is one reason why we did not include further checking, such as a checksum of kilobytes of the stack.

Although this signature recording and checking mechanism has proven to have low overhead and high reliability, it is not foolproof. A sequence of code could be generated that incremented or decremented memory in a loop as a loop counter, with all other registers and stack remaining the same across iterations. In this case, we may trigger a false positive match on the first iteration rather than a subsequent iteration. An enhanced version of the signature detection could be developed to include results of memory operations when no registers change, but in practice, we have yet to encounter this false positive situation.

4.5. Merging Results

Because SuperPin slices an application into separate processes with their own copy of Pin and the Pintool, the data a Pintool records will only be local to its slice. A *merge* function must be called to combine the output of the last completed slice into a collective total. To aid in determinism, the merge function is called *in slice order*.

If we are counting instructions or profiling dynamic instruction types, the merge function will simply add each local value to a running total using a shared memory region. When the program completes, a `fini` function is called which displays the results from shared memory. As another example, if we are tracing instructions, the slice output will be buffered, then appended to the output during merging.

SuperPin is not designed to handle cases where the machine state is abnormally changed while the application executes. However, with clever design, many tools can be converted to a SuperPin tool. Though some tools may depend on data from a previous slice, this can be resolved at merge time. The general steps are as follows:

1. Assume previous slice ends with a particular state and record data accordingly.
2. Compare the actual ending state of the previous slice with the assumptions made. A slice may need to keep track of extra data just for this purpose.

3. Reconcile the differences between the assumptions and reality during the merge routine.

In the next section, we will illustrate how this can be accomplished with a data cache tool.

5. SuperPin Applications

We have extended Pin's API to allow tool writers to take advantage of SuperPin's features and even affect its behavior. This API is included as a standard feature in the latest toolkit on the Pin web site [1]. We describe the new SuperPin API in this section and provide details of two sample tools – an instruction counter tool and a data cache tool.

SuperPin API Pin's API has been extended to provide full support controlling and accessing SuperPin. The following calls are available to a tool writer:

- `SP_Init(fun)`: Initialize the SuperPin system. Must be called before `PIN_StartProgram` and after `PIN_Init`. Pass it function *fun* which is used to reset local statistics. Returns true if using SuperPin.
- `SP_AddSliceBeginFunction(fun,val)`: Calls function *fun* immediately after a new SuperPin slice has been created.
- `SP_AddSliceEndFunction(fun,val)`: Calls function *fun* right before a SuperPin slice terminates.
- `SP_EndSlice()`: Tool instructs SuperPin to terminate this slice immediately.
- `SP_CreateSharedArea(localData, size, autoMerge)`: Allocate shared memory region of *size* length and return a pointer to that region if successful and using SuperPin. Otherwise return localData. AutoMerge indicates if and how this region is automatically merged (e.g. addition).

An example of a SuperPin tool that uses the `SP_EndSlice` function is the Shadow Profiler Pintool [18], which performs sampled profiling via instrumented timeslices, achieving lower overhead than is attainable via full instrumentation.

SuperPin Switches To use SuperPin, a tool must be written using the presented API, and Pin must be executed with the switch `-sp 1`. The following command-line switches are also available for controlling SuperPin's behavior.

- *-spmsec {value}*: Number of milliseconds for a timeslice (default: 1000)
- *-spmp {value}*: Maximum number of running slices (default: 8)
- *-spsysrecs {value}*: Maximum number of system call records per slice, or 0 to disable system call recording (default: 1000)

5.1. Icount SuperTool

Icount is a simple tool that counts the number of instructions executed by an application and all of its shared libraries. Two versions of the traditional `icount` pintool are shipped with Pin. The first version, `icount1`, instruments the application at the granularity of an instruction. A call is inserted after every instruction that increments a global counter. An optimized version of this Pintool is called `icount2`, which operates at a basic-block granularity. At the end of every basic block, a call is inserted that increments a global counter by the number of instructions in that basic block.

Figure 2 presents the SuperPin version of *icount2*. The tool required a few modifications in order to use Super-Pin's features (highlighted). First, starting from the top, we have added a pointer, `*sharedData`. This pointer refers to memory that is either shared across all slices or local to each slice, depending upon whether the tool is run in SuperPin or traditional Pin mode. Second, we have added a `ToolReset` function. This function resets the local instruction count at the start of each slice.

Next, in the `main` function, we have added a call to `SP_Init` and have passed the tool reset function, `ToolReset`. We then call `SP_CreateSharedArea` passing along a pointer to our local data and the size of the shared memory region. If we are not in SuperPin mode, the `sharedData` pointer will be assigned to the local pointer we pass to that function.

Finally, we call `SP_AddSliceEndFunction` and pass it our manual `Merge` function. We are using a manual merge function instead of an automatic merge function for illustrative purposes. In this case, `Merge` simply adds the local instruction count value, `icount` to the value pointed by the shared pointer. While it is possible to display the slice count and running total at this time, we have chosen to suppress our output until program completion.

5.2. Data Cache SuperTool

Merging the output of an instruction counting tool turned out to be straightforward, but that may not always be the case. Other instrumentation tasks may have dependences between the timeslices. These dependences don't have to preclude the use of SuperPin, however. Here we describe how a data cache simulator, such as `SimpleExamples/dcache.cpp` as provided in the Pin toolkit, can be modified to be amenable to SuperPin (using the procedure outlined in Section 4.5). Consider the case of a direct-mapped data cache. We assume that the first access in a slice will be a hit. Next, upon the first access, we increment our hit counter, but also make a special record of the line address containing the assumed hit. Then, when

Icount SuperTool
```
#include <iostream>
#include "pin.H"

UINT64 icount = 0;
// points either to shared or local statistical data
UINT64 *sharedData;

VOID docount(INT32 c) { icount += c; }

// NEW: Clears slice local data
VOID ToolReset(UINT32 sliceNum) {
  icount = 0;
}
// NEW: Merge local to shared data
VOID Merge(INT32 sliceNum, VOID *v) {
  *sharedData += icount;
}
VOID Fini(INT32 code, VOID *v) {
  // use sharedData pointer now
  std::cout << "Total Count: " << *sharedData << endl;
}
VOID Trace(TRACE trace, VOID *v) {
  for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl);
       bbl = BBL_Next(bbl))
    INS_InsertCall(BBL_InsHead(bbl), IPOINT_BEFORE,
                   (AFUNPTR)docount, IARG_UINT32,
                   BBL_NumIns(bbl), IARG_END);
}
int main(INT32 argc, CHAR **argv) {
  PIN_Init(argc, argv);

 /* BEGIN SuperPin */
  SP_Init(ToolReset);
  sharedData = (UINT64 *)SP_CreateSharedArea(&icount,
                         sizeof(icount), 0);
  SP_AddSliceEndFunction(Merge, 0);
 /* END SuperPin */

  TRACE_AddInstrumentFunction(Trace, 0);
  PIN_AddFiniFunction(Fini, 0);

  PIN_StartProgram();  // Never returns
  return 0;
}
```

Figure 2. SuperPin version of icount2

the slice completes, we compare the line of our first access with the final cache state of the previous slice. If they do not match, we subtract the assumed hit and add a miss to our record. We then add the hits and misses to the shared merged record.

6. Performance Comparison

We evaluated the performance of SuperPin against Pin in traditional execution mode and against the native application execution time. We used `icount1` tool, which instruments each individual instruction with a counter increment, and `icount2`, which performs the instrumentation at a basic block granularity. While the output of both tools will be identical, the `icount2` tool will have much lower overhead. We ran both tools on the SPEC2000 suite of benchmarks on an 8-way multiprocessor using 2.2 GHz Intel®Xeon®MP processors. In SuperPin mode, the maximum number of slices was limited to 8 (one per processor),
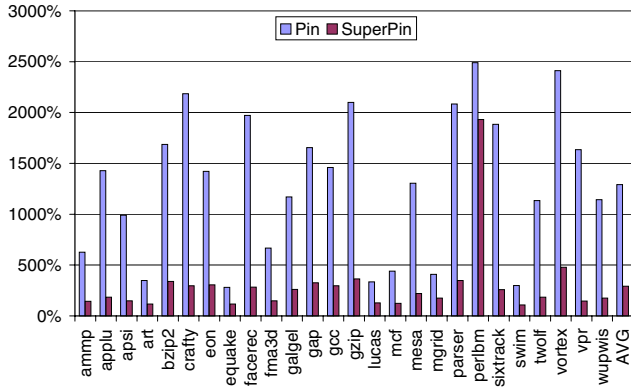
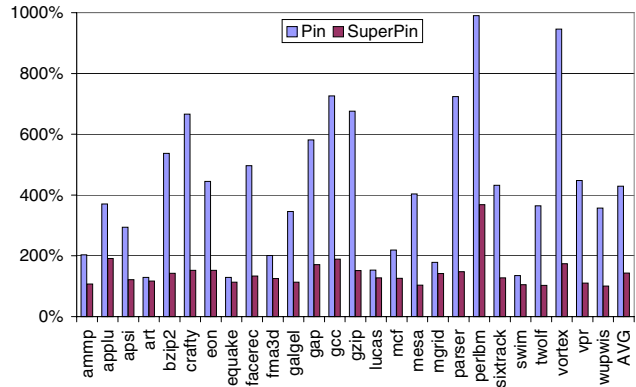**Figure 3. icount1: Pin and SuperPin performance relative to native run**

Figure 5. icount2: Pin and SuperPin performance relative to native run
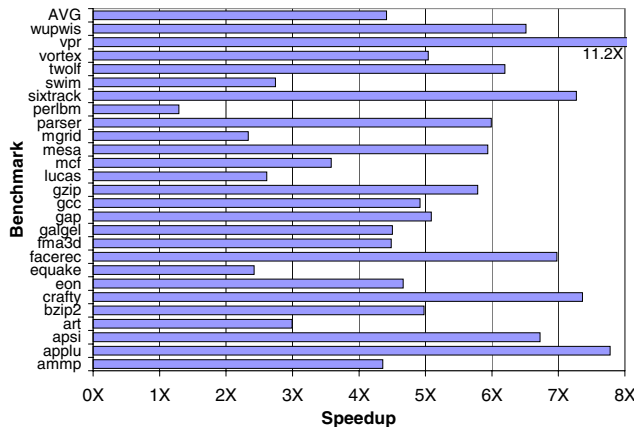

Figure 4. icount1: SuperPin speedup over Pin

and the timeslice interval was set at 2 seconds (we later vary the processor count and timeslice interval).

The results of running the `icount1` tool are shown in Figure 3. We present the normalized execution time of the tool in traditional mode and SuperPin mode (relative to native). On average, there was about a 12X slowdown in traditional mode. Figure 4 shows the corresponding speedup of SuperPin over Pin for `icount1`, where SuperPin outperforms Pin by 3 to over 7 times. For `icount1`, the tool is *instrumentation-limited*. There is not enough parallelism available to overcome the work required by the `icount1` tool. Therefore, the SuperPin speedup usually will not exceed 8X unless significant cache locality benefits arise.

On the other hand, for the `icount2` tool, there is enough parallelism in most cases for the application to come close to real-time performance, as we can see from Figure 5. Compared to native execution, we saw a 25% average slowdown while running with SuperPin instrumentation (rang-

ing from 7% to just under 100% slowdown). The variation is primarily a result of application behavior, which affects SuperPin overhead both directly and indirectly. The duration of the application affects the pipeline delay. It becomes difficult to achieve slowdowns under 25% for applications with shorter execution times.

6.1. Timeslice Variation Effects

Next, we explored the effect of changing the timeslice interval. Figure 6 shows the run time of the `gcc` benchmark as we varied the timeslice interval from 0.5–4 seconds. We chose `gcc` because its large code footprint and higher overhead best illustrate the effects of changing the timeslice interval. We also restricted the input set to one input to properly reflect the pipeline delay. This resulted in a native run time of nearly 100 seconds. Figure 6 breaks the results down into the run time overhead components of forking and other losses, stalls within the master application to avoid exceeding maximum number of slices, and the pipeline delay after the master application has finished.

The figure indicates that as the timeslice size increases, the overhead due to forking and other inherent costs with running multiple processes decreases. Also, the master does not sleep as much because the slices do not have to compile as frequently. On the other hand, the pipeline delay will increase. However, the benefits of the lower overhead outweigh the pipeline delay for `gcc`, and a net runtime reduction is seen which levels off at higher time slice sizes.

6.2. Parallelism Variation Effects

The number of virtual processors available to execute slices has a direct relationship to the performance of SuperPin. This is demonstrated in Figure 7, where the maximum number of slices is varied from 1 to 16 for the `gcc`
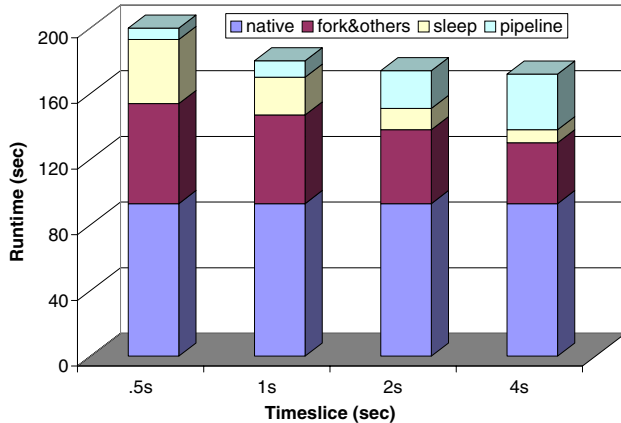
Figure 6. Timeslice interval variation for `gcc`

Figure 7. Impact of available processor parallelism on performance

benchmark. The machine used is only 8-way SMP, but with hyperthreading enabled, it was extended to 16 virtual processors. With only 2 processors, we saw little benefit after the overhead, but performance improved dramatically until we reached the limit of physical processors. When using 16 processors, the master application does not wait at all – it will fork as quickly as possible. However, the hyperthreading effect of sharing a processor reduces the master application's performance, so it is not quite real time. In this case, the execution is said to be *application limited*.

6.3. Overhead

There are three main reasons why our performance approaches, but does not reach the performance of a single processor native run. These are pipeline delay, compilation slowdown, and master application slowdown.

1. Pipeline Delay. As explained in Section 3, the design of SuperPin pipelines timeslices of an application. As a result, there will be a theoretical minimum slowdown based on the duration of a timeslice in order for the final timeslice to complete.

2. Compilation Slowdown. Because each slice has its own copy of the code cache, and it starts in a clean state when each slice begins, the compilation engine is doing much more work. This slowdown is significant only in *instrumentation-limited* runs. Otherwise, the extra parallelism hides the compilation overhead.

3. Master Application Slowdown. The *running environment* may degrade the performance of the master process in some instances. This can be caused by the operating system, the hardware, or SuperPin overhead. The subcomponents of this overhead are:

    Ptrace Overhead   The master application is controlled via `ptrace`, and after every system call the
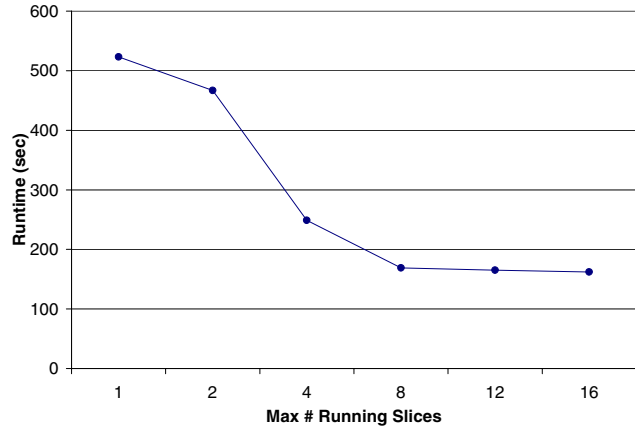
master application re-enters the VM for bookkeeping purposes. This overhead has been measured to be less than a few tenths of a percent.

Fork Overhead   When a slice forks, it creates a copy of the application space via copy-on-write page faults. Every time the application or a slice touches a clean page, the page must be copied.

Operating System Scheduling   Poor scheduling by the OS can cause swap and cache thrashing.

Hyperthreading   If the master application is forced to share its CPU with another slice or external process, this will impact performance.

SMP Scalability Issues   Running on all processors taxes the memory and other subsystems. As a result, each process on the individual CPU will run slower. This has been verified by loading a system with the same benchmark running natively. It will run slower than running a single instance with no other load on the system. In fact, many SuperPin *application-limited* runs are very close to the time under this scenario.

7. Conclusion

We have introduced a novel approach to improving the performance of dynamic instrumentation. Our technique works by dividing the program into many timeslices, executing those timeslices with instrumentation in parallel, and merging the results.

We performed a complete implementation of our approach in Pin, and made many important design decisions en route. One interesting feature is an effective signature detection mechanism to trigger the end of an instrumentation

slice. A full performance evaluation then illustrated that the overhead came in several forms: pipeline delay, compilation slowdown, and master application slowdown.

For a certain class of instrumentation tasks that are amenable to parallelization and merging, the SuperPin approach provides significant performance improvements, often an order of magnitude improvement over the widely used method for dynamic instrumentation.

8. Future Work

SuperPin exhibits significant speedup over non-parallelized instrumentation, however, there is still room for improvement. Section 6 described some of the overheads involved and our future work is to continue to optimize these overheads.

Solving the problem of pipeline delay is challenging, but we envision a mechanism that could automatically throttle the timeslice duration. Ideally, we would be able to decrease the timeslice size toward the end of application execution. We are investigating realistic approximations of that goal.

The best approach for dramatically reducing the compilation overhead may be to share the code cache across all timeslices via shared memory. This may add a little extra overhead by performing extra consistency checks from other slices, but we feel that the reduction in overhead will outweigh the costs.

Finally, we would like to provide multithreading support to our implementation. Though this will require deterministic replay of threads, we are working with experts in that area to accomplish this task.

References

[1] Pin website. http://pin.webhop.org.
[2] E. R. Altman, M. Gschwind, S. Sathaye, S. Kosonocky, A. Bright, J. Fritz, P. Ledak, D. Appenzeller, C. Agricola, and Z. Filan. BOA: The architecture of a binary translation processor. *IBM Research Report RC 21665*, Dec. 2000.
[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Programming Language Design and Implementation*, pages 1–12, 2000.
[4] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach. IA-32 execution layer: a two-phase dynamic translator designed to support ia-32 applications on itanium-based systems. In *36th Intl. Symposium on Microarchitecture*, pages 191–201, Dec. 2003.
[5] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Code Generation and Optimization*, pages 265–275, Mar. 2003.
[6] H. Chen, J. Lu, W. Hsu, and P.-C. Yew. Continuous adaptive object-code re-optimization framework. In *Ninth Asia-Pacific Computer Systems Architecture Conference*, pages 241–255, 2004.

[7] W.-K. Chen, S. Lerner, R. Chaiken, and D. Gillies. Mojo: A dynamic optimization system. In *4th Workshop on Feedback-Directed and Dynamic Optimization*, pages 81–90, 2000.
[8] C. Cifuentes, B. Lewis, and D. Ung. Walkabout - a retargetable dynamic binary translation framework. Technical Report TR2002-106, Sun Microsystems Labs, Jan. 2002.
[9] D. Deaver, R. Gorton, and N. Rubin. Wiggins/redstone: An on-line program specializer. In *IEEE Hot Chips XI*, 1999.
[10] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. The transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Code Generation and Optimization*, pages 15–24, Mar. 2003.
[11] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: A new run-time control point. In *35th Intl. Symposium on Microarchitecture*, pages 257–268, 2002.
[12] K. Ebcioglu and E. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *24th Intl. Symposium on Computer Architecture*, pages 26–37, June 1997.
[13] R. J. Hookway and M. A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, pages 3–12, Feb. 1997.
[14] H.-S. Kim and J. E. Smith. Dynamic binary translation for accumulator-oriented architectures. In *Code Generation and Optimization*, pages 25–35, Mar. 2003.
[15] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, San Francisco, Aug. 2002.
[16] B. C. Le. An out-of-order execution technique for runtime binary translators. In *8th Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 151–158. ACM Press, 1998.
[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapareddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Programming Language Design and Implementation*, Chicago, IL, June 2005.
[18] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. V. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *Code Generation and Optimization*, Mar. 2007.
[19] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic Notes in Theoretical Computer Science*, 89(2), 2003.
[20] H. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *Automated and Algorithmic Debugging*, pages 119–132, 1995.
[21] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Code Generation and Optimization*, pages 243–254, Mar. 2005.
[22] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. Davidson, and M. L. Soffa. Reconfigurable and retargetable software dynamic translation. In *Code Generation and Optimization*, pages 36–47, Mar. 2003.
[23] D. Ung and C. Cifuentes. Optimising hot paths in a dynamic binary translator. In *2nd Workshop on Binary Translation*, Oct. 2000.

IEEE
COMPUTER
SOCIETY