Controlling Program Execution through Binary Instrumentation

Heidi Pan and Krste Asanović
Massachusetts Institute of Technology
{xoxo, krste}@csail.mit.edu

Robert Cohn and Chi-Keung Luk
Intel Corporation
{robert.s.cohn,chi-keung.luk}@intel.com

Abstract

Binary instrumentation has been widely used to observe dynamic program behavior, but current binary instrumentation systems do not allow the tool writer to alter the program execution path. This paper introduces some simple and general mechanisms for a binary instrumentation infrastructure to provide control over the application's execution path, allowing tools to replay or skip parts of the application, and to start or switch between threads. Specifically, the technique provides the following three functionalities for both single-threaded and multi-threaded applications: (1) checkpointing the execution state, (2) resuming execution at a checkpoint, and (3) starting execution at an arbitrary point in the program with a specified architectural state. We describe our implementation of these functionalities in Pin, a dynamic binary instrumentation infrastructure from Intel [5]. We demonstrate the usefulness of our mechanism by describing several binary instrumentation tools that have been built using this interface, including a transactional memory model and a thread scheduler.

1 Introduction

Proposals for new computer architecture features are usually evaluated through extensive simulation. Many simulators are divided into a front-end functional simulator that executes instructions to model their effect on the architectural state of the machine, and a back-end performance model which takes instruction information from the functional simulator and calculates expected behaviour of a proposed microarchitecture. For simpler microarchitectures, no feedback is required from the back-end performance model to the front-end functional simulator, and the front-end could be replaced with a stored trace of program execution. Many modern microarchitectures, however, are considerably more complex, and accurate execution-driven simulation requires that the back-end control the execution of the front-end. For example, with speculative execution, the back-end directs the front-end to execute down the predicted path until the branch is resolved, at which point the front-end should restore the architectural state at the point of the misprediction before continuing along the correct path.

Binary instrumentation is a powerful technique for implementing architectural simulators, whereby an application executable binary is translated into a new version with instrumentation code added [5, 7, 8, 1, 3]. For example, the binary can be instrumented to insert code before every memory instruction to simulate the effect of the memory reference on the cache. Binary instrumentation provides a much faster implementation of the front-end functional simulator compared with conventional instruction set interpreters, as the instrumented binary runs natively with no interpretive overhead. Another important advantage is that binary instrumentation can leverage the host machine environment to provide compilation tool chains, system call interfaces, and user-level libraries. However, current binary instrumentation infrastructures do not allow instrumentation code to alter the application's execution path, effectively restricting their use to providing feeders for trace-driven simulators.

In this paper, we present an extension to the Pin dynamic binary instrumentation infrastructure to allow the program path of execution to be controlled by instrumentation code. We describe a new API that hides details of the underlying dynamic binary translation system from the instrumentation code, and show how this can be used to construct efficient execution-driven simulators for both complex uniprocessors and multiprocessor systems.

2 Pin Extensions

In this section, we describe extensions to the Pin API to allow tools to checkpoint or execute at arbitrary points in a program. The full program state encompasses register, memory, and OS state. The extensions for checkpointing and execution control only manipulate registers and the instruction pointer. Memory and OS state checkpointing can be performed using the existing instrumentation API, as described in Section 3.2.

Object(s) / Function(s)	Description
CONTEXT* IARG_CONTEXT	current application architectural state
ADDRINT PIN_GetContextReg(CONTEXT*, REG)	get the value of the register in the given context
VOID PIN_SetContextReg(CONTEXT*, REG, ADDRINT)	set the value of the register in the given context
VOID PIN_ExecuteAt(CONTEXT*)	execute at the given program context

Table 1. Context API

Object(s) / Function(s)	Description
CHECKPOINT* IARG_CHECKPOINT	current processor state
VOID PIN_SaveCheckpoint(CHECKPOINT*, CHECKPOINT*)	save IARG_CHECKPOINT for later use
VOID PIN_Resume(CHECKPOINT*)	resume execution at the given checkpoint

Table 2. Checkpoint API

```
/* binary instrumentation tool - instrumentation */
// following instrumentation causes application to
// continue at Func() instead of continuing normally
if (INS_Address(ins) == 0x400000)
{
    INS_InsertCall(ins, IPOINT_BEFORE, AFUNPTR(JumpToFunc),
        IARG_CONTEXT, IARG_END);

}

/* binary instrumentation tool - analysis routine */
void JumpToFunc(CONTEXT* ctxt)
{
    PIN_SetContextReg(ctxt, REG_INST_PTR, Func);
    PIN_ExecuteAt(ctxt);
}

/* application code */
int Func() { ...  }
```

Figure 1. Changing application control flow with `PIN_ExecuteAt`.

There are many challenges in controlling the program execution path in a dynamic binary instrumentation infrastructure, where instrumentation code is inserted "on-the-fly" and cached in an instrumented code cache. During the dynamic translation process, the original program instructions are rewritten and registers are reallocated to improve performance. When the instrumentation code needs to change the control flow, it cannot simply change the instruction pointer. The instruction pointer must be redirected to the appropriate location in the code cache, or used to start translating a new trace of instructions into the code cache.

To hide details of the underlying translation system, we represent the application's architectural state with the `CONTEXT` object. Table 1 lists the interface for obtaining the current application architectural state, reading and writing individual register values, and executing at a different point in the program with a new context.

The pseudocode shown in Figure 1 demonstrates the simplicity of changing the program execution path in Pin using our context interface. In this example, the tool inserts a call to `JumpToFunc` before an instruction at address 0x400000. By listing `IARG_CONTEXT`, it requests that the current application context be passed to `JumpToFunc`. Immediately before executing the instruction at address 0x400000, it will call JumpToFunc. The function JumpToFunc takes the context argument, changes the instruction pointer to point to the Function `Func`. At this point, it is also possible to read and write the values of the application context using the `PIN_GetContextReg` and `PIN_SetContextReg` functions. Finally, it calls `PIN_ExecuteAt` which transfers control to an instrumented version of `Func`.

The context interface is very general, but may require some dynamic compilation and conversion between Pin's internal state and the architectural state of the program. Checkpointing can be implemented by saving and restoring the application architectural state using the context interface, but it is more efficient to save and restore the architectural state of the underlying instrumentation system. We provide a more streamlined checkpoint/restore mechanism when the tool only needs to save and restore an application state without reading or writing any of the individual registers.

The checkpoint interface is listed in Table 2. Instead of `CONTEXT`, we save the state in a `CHECKPOINT` object. Figure 2 shows a simple example of a tool checkpointing the function `Foo` and resuming later at the checkpoint, essentially executing `Foo` twice. `IARG_CHECKPOINT` must be saved by the tool using `PIN_SaveCheckpoint`, since its lifetime only lasts until `CheckpointFoo` returns.

3 Tools

We demonstrate the importance of providing control over the application's execution path by describing two

```
/* application */
int main(int, char**)
{
    Foo(1);
    Bar(2);
}

/* binary instrumentation tool - analysis routines */
CHECKPOINT chkpt;
BOOL replayed = false;
void CheckpointFoo(CHECKPOINT* _chkpt)
{
    PIN_SaveCheckpoint(_chkpt, &chkpt);
}

void RevertBackToFoo()
{
    if (!replayed)
    {
        PIN_Resume(&chkpt);
        replayed = true;
    }
}

/* binary instrumentation tool - instrumentation */
if (RTN_Address(rtn) == Foo)
{
    RTN_InsertCall(rtn, IPOINT_BEFORE,
        AFUNPTR(CheckpointFoo), IARG_CHECKPOINT, IARG_END);

}

if (RTN_Address(rtn) == Bar)
{
    RTN_InsertCall(rtn, IPOINT_BEFORE,
        AFUNPTR(RevertBackToFoo), IARG_END);

}
```

Figure 2. Executing Foo(1) Twice Using `PIN_Resume`.

multiprocessor simulator tools built upon this feature.

3.1 User-Level Thread Library

In this section, we present a user-level thread library, implemented as a binary instrumentation tool, that enables architectural simulators to schedule threads based on detailed processor, cache, memory, and interconnect models.

The relative rate of execution of different threads drastically affects the measured sharing patterns and coherence traffic. If a traditional instrumentation system is used for a multiprocessor, with each application thread translated separately on a different host OS thread, then the relative rate of thread execution is dependent on the host's memory system, processor count, and OS scheduler. In addition, the extra instrumentation code may introduce non-uniform latency across the different threads, further distorting the threads' relative rates. In contrast, our mechanism provides control over each thread's execution, allowing instrumentation code to govern the thread interleaving based on multiprocessor performance models to more accurately explore future multiprocessor designs.

To gain full control of thread execution, the thread library runs multithreaded applications on a single host OS thread, managing user-level threads unknown to the operating system and Pin. The library consists of two main components: the thread manager and the thread scheduler. The thread manager implements the thread functionalities needed by the application, such as thread creation and cancellation, manipulation of mutexes and condition variables, thread local storage, and thread-safe memory allocation and IO operations. We adopt the pthread library interface to maintain compatibility with existing applications, toolchains, and libraries (e.g. glibc, openmp). In our implementation, we relink the application with a dummy pthread library, then instrument all pthread routines called by the application to execute code from our thread manager.

The thread scheduler maintains a queue of threads and runs them one at a time. We have implemented a simple round-robin scheduler, though our thread library is capable of supporting various types of scheduling policies. Based on feedback from the detailed architectural timing model, the scheduler can control the relative rate of the threads by deciding how long each thread can run before being swapped out. The scheduler is also responsible for suspending threads when they are joining an unfinished thread, spinning on a lock, or waiting for a condition.

The scheduler relies on our context and checkpoint interface for two important mechanisms: starting a new thread and context switching between threads. Figure 3 illustrates how to start a new thread. The instrumentation call passes the current context along with `pthread_create`'s arguments to the analysis routine, `StartThread`. When the application calls `pthread_create`, we create a pthread object with the given attributes, allocating the thread's stack in the process. The new thread inherits its parent thread's context, but sets the program counter to its start routine and the stack pointer to the newly allocated stack, and pushes the initial argument onto the stack. Calling `PIN_ExecuteAt` with the modified context starts the actual execution of the thread.

To context switch between threads, we simply save the current thread's checkpoint and resume at the next thread's saved checkpoint, as shown in Figure 4. The frequency of context switching is controlled by granularity of instrumentation. For example, to context switch at every instruction, we would insert a call to the `ContextSwitch` analysis routine before every instruction.

3.2 Transactional Memory

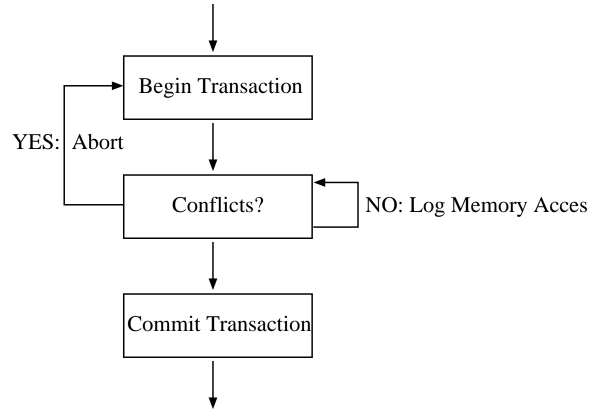
In traditional multiprogramming systems, locks are used to enforce data dependencies and timing constraints

```
/* application */
pthread_create(thread, attr, startrtn, arg);

/* binary instrumentation tool - instrumentation */
if (RTN_Address(rtn) == pthread_create)
{
    RTN_InsertCall(rtn, IPOINT_BEFORE,
        (AFUNPTR)StartThread, IARG_CONTEXT,
        IARG_ARG0, IARG_ARG1, IARG_ARG2,
        IARG_ARG3, IARG_END);

}

/* binary instrumentation tool - analysis routine */
void StartThread(CONTEXT* ctxt,
    pthread_t* thread, pthread_attr_t* attr,
    void*(*startrtn)(void*), void* arg)
{
    th = new Pthread(attr);
    *(th.sp--) = arg;
    PIN_SetContextReg(ctxt, REG_STACK_PTR, th.sp);
    PIN_SetContextReg(ctxt, REG_INST_PTR, startrtn);
    PIN_ExecuteAt(ctxt);
}
```

Figure 3. Starting a New Thread Using `PIN_ExecuteAt`

```
/* binary instrumentation tool - analysis routine */
void ContextSwitch(CHECKPOINT* chkpt = IARG_CHECKPOINT)
{
    PIN_SaveCheckpoint(chkpt, currentthread.chkpt);
    PIN_Resume(nextthread.chkpt);
}
```

Figure 4. Context Switching Using `PIN_Resume`




```
/* binary instrumentation tool - instrumentation */
if (RTN_Address(rtn) == XBEGIN)
{
    RTN_InsertCall(rtn, IPOINT_BEFORE,
        (AFUNPTR)BeginTransaction,
        IARG_THREAD_ID, IARG_CHECKPOINT, IARG_END);

}
if (RTN_Address(rtn) == XEND)
{
    RTN_InsertCall(rtn, IPOINT_BEFORE,
        (AFUNPTR)CommitTransaction,
        IARG_THREAD_ID, IARG_END);

}
if (INS_IsMemoryWrite(ins))
{
    INS_InsertCall(ins, IPOINT_BEFORE,
        (AFUNPTR)DetectConflict, IARG_BOOL, true,
        IARG_THREAD_ID, IARG_MEMORYWRITE_EA,
        IARG_MEMORYWRITE_SIZE, IARG_END);

}
if (INS_IsMemoryRead(ins))
{
    INS_InsertCall(ins, IPOINT_BEFORE,
        (AFUNPTR)DetectConflict, IARG_BOOL, false,
        IARG_THREAD_ID, IARG_MEMORYREAD_EA,
        IARG_MEMORYREAD_SIZE, IARG_END);

}

/* binary instrumentation tool - analysis routines */
CHECKPOINT chkpt[NTHREADS];
LOG log[NTHREADS];
void BeginTransaction(ADDRINT th, CHECKPOINT* _chkpt)
{
    PIN_SaveCheckpoint(_chkpt, &chkpt[th]);
}
void CommitTransaction(ADDRINT th)
{
    /* discard chkpt[th] and log[th] */
}
void DetectConflict(BOOL iswrite, ADDRINT th,
    ADDRINT addr, ADDRINT len)
{
    if ( /* in a transaction */ )
    {
        if ( /* conflict */ )
        {
            /* restore the memory with log[th] */
            PIN_Resume(&chkpt[th]);
        }
        else
        {
            /* record this memory access in log[th] */
        }
    }
}
```

Figure 5. Transactional memory model using the checkpoint interface.

between the various threads. However, locks are difficult to reason about, often leading to unwanted race conditions, priority inversion, or deadlock. Therefore, a recent wave of architectural research projects ([2, 4, 6]) are exploring transactional memory systems as an alternative synchronization mechanism to locks.

A transaction is a sequence of instructions that either commits or aborts. Once a transaction commits, all of its instructions appear to have executed atomically, but no programmer-visible state is altered if a transaction is aborted. In practice, transactions may execute in parallel, optimistically assuming that they do not touch the same data. As long as no transactions modify any memory locations accessed by another concurrent transaction, the transactions commit successfully. However, if a conflict is detected between two transactions, one of the transactions must abort immediately and restore memory before the other proceeds.

Figure 5 depicts a basic transactional memory model using the checkpoint interface described in the previous section. We are interested in instrumenting three types of events – the beginning of the transaction, the end of the transaction, and all memory accesses in between. A thread must checkpoint its program state upon entering a transaction, in case it needs to abort the transaction and roll back execution. Since Pin only checkpoints the processor state, the tool is responsible for checkpointing the memory state. Fortunately, instead of having to save the entire memory image, the tool only has to remember the original values of memory locations modified by the transaction. This incurs minimal overhead, since the tool must already log all the memory instructions inside transactions to detect conflict between concurrent transactions. By examining other threads' logs, a thread can ensure that it is not modifying a memory location accessed by another transaction or accessing a memory location modified by anoter transaction. If a conflict is detected and the thread is chosen to abort, it must use its log to restore all the memory locations it has modified, and revert back to its saved checkpoint to retry the transaction. If the thread completes the transaction without detecting any conflicts, it can effectively commit its changes by discarding the checkpoint and log.

Through the checkpoint interface, we provide a basic mechanism in the binary instrumentation infrastructure to speculatively execute and abort transactions. Extricated from the details of how to undo program execution, architects can easily build complex transactional memory models exploring different conflict detection algorithms, abort policies (choosing which transaction to abort to provide fairness and prevent deadlock), and backoff strategies (to guarantee forward progress of aborted transactions). In addition, the same mechanism can enable development of other speculative architectural simulator tools, such as branch and value predictors.

4 Implementation

Both the application and the tool should observe the original program behavior and state, and Pin is responsible for preserving this transparency. The baseline Pin implementation transitions between the application and the tool through a *bridge routine*, which consists of generated code to save all caller-saved registers and set up analysis routine arguments. To provide the tool with `IARG_CONTEXT`, we add extra code at the beginning of the bridge to capture the application's architectural state. Each application register is read from its allocated register or spill location in memory. Special care must be taken to preserve the architectural state until it is entirely saved, since the act of capturing the context may inadvertently alter register or memory values. For example, we must calculate addresses using the load-effective-address instruction rather than a simple add instruction before we save the eflags register, since the latter has a side effect of changing condition codes. To restore the context for `PIN_ExecuteAt`, the application IP is translated into the code cache IP, and the architectural register values are written back to the appropriate registers and memory spill locations. If the application IP corresponds to code that has not yet been instrumented, Pin dynamically compiles the code and restores the register values using the same register mapping.

Despite the simplicity of the context interface, it is not necessary to incur the performance penalty of translating between the application and runtime state if the tool does not need to manipulate the individual registers. `IARG_CHECKPOINT` is created at the beginning of the bridge to capture the runtime architectural and spilled register values, so no register translation is necessary. It also records the code cache address rather than the original application IP. Since `PIN_Resume` reverts back to a previous point in execution stored in the code cache, it never needs to invoke dynamic generation of new code.

5 Evaluation

In this section, we will try to quantify the cost of changing the program execution path. Saving and restoring the entire register state is expensive. The cost is even higher when dealing with a `CONTEXT`, since we need to translate between the application and the infrastructure state, and may also need to dynamically generate code when jumping to a new point in the program. Table 3 compares the cost of context switching using `PIN_Resume` versus `PIN_ExecuteAt`. We

Baseline	PIN_Resume	PIN_ExecuteAt
35.2s	1469.3s	6351.3s

Table 3. Cost of Context Switching Using PIN_Resume vs PIN_ExecuteAt

start with a simple baseline tool that instruments every instruction to increment an instruction count while running gzip on a large text file. We extend the analysis routines to obtain the current state at each application instruction, either through IARG_CHECKPOINT or IARG_CONTEXT. The tool then resumes in place by either calling PIN_Resume or PIN_ExecuteAt using the respective IARG. Although the program execution path is not altered, we can still measure the overhead of a single context switch by taking the timing difference from the baseline divided by the total number of instructions executed (527,276,783 in this case). More importantly, we show that context switching with PIN_ExecuteAt is more than 4 times as expensive as with PIN_Resume, illustrating the tradeoff between a more user-friendly interface and a more efficient implementation.

The tool writer can reduce the cost of context switching by using conditional instrumentation. Even if the tool can dynamically determine in the analysis routine that context switching is unnecessary, it has already paid the cost of creating the IARG_CHECKPOINT object. Instead, the tool can create two separate analysis routines using PIN_InsertIfCall and PIN_InsertThenCall. The "if" analysis routine determines whether the "then" analysis routine should be called. Figure 6 illustrates the simulation time savings of conditionally passing IARG_CHECKPOINT. The baseline tool (represented by the dotted line) passes the IARG at every application instruction, but does nothing beyond incrementing the instruction count inside the analysis routine. The remaining tools increment the instruction count inside the if analysis routine, and passes the IARG to a dummy then analysis routine every n instructions. When passing the IARG more frequently than every 20 instructions, the conditional instrumentation actually performs worse than the baseline since it incurs the cost of two analysis routines per instruction. However, the cost of the extra analysis routine is outweighed by the saving of not passing the IARG as the frequency decreases.

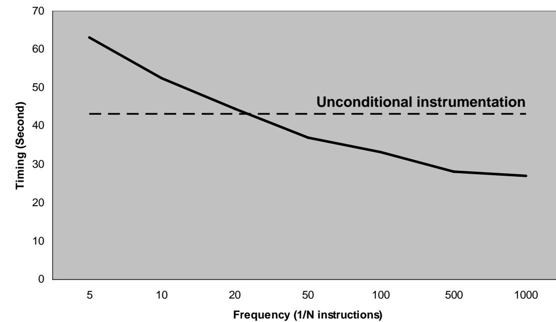


Figure 6. Cost of Conditionally Obtaining IARG_CHECKPOINT at Different Frequencies

6 Conclusion

We have presented a binary instrumentation technique to provide control over the application's execution path. Our technique enables architectural simulators to model speculative systems and control the thread interleaving of multithreaded applications. We have demonstrated the utility of our technique with a sample multiprocessor thread scheduler and a transactional memory model, and evaluated the performance costs of saving and restoring state. The checkpointing and execution control extensions, as well as the two tools described in this paper are distributed with the current release of Pin, downloadable from http://rogue.colorado.edu/Pin.

References

[1] *Assembly Language Programmer's Guide (Pixie)*. MIPS Computer Systems, Inc., 1986.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA*, 2005.

[3] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *International Journal of High Performance Computing Applications*, pages 317–329, 2000.

[4] Lance Hammond et al. Transactional memory coherence and consistency. In *ISCA*, 2004.

[5] Chi-Keung Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[6] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA*, 2005.

[7] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *PLDI*, 1994.

[8] Emmett Witchel and Mendel Rosenblum. Embra: Fast and flexible machine simulation. In *SIGMETRICS*, 1996.