# Framework for Instruction-level Tracing and Analysis of Program Executions

Sanjay Bhansali Wen-Ke Chen Stuart de Jong Andrew Edwards Ron Murray
Milenko Drinić Darek Mihočka Joe Chau

Microsoft Corporation, One Microsoft Way, Redmond, WA
{sanjaybh,wenkec,sdejong,andred,ronm,mdrinic,darekm,joechau}@microsoft.com

## Abstract

Program execution traces provide the most intimate details of a program's dynamic behavior. They can be used for program optimization, failure diagnosis, collecting software metrics like coverage, test prioritization, etc. Two major obstacles to exploiting the full potential of information they provide are: (*i*) performance overhead while collecting traces, and (*ii*) significant size of traces even for short execution scenarios. Reducing information output in an execution trace can reduce both performance overhead and the size of traces. However, the applicability of such traces is limited to a particular task. We present a runtime framework with a goal of collecting a complete, machine- and task-independent, user-mode trace of a program's execution that can be re-simulated deterministically with full fidelity down to the instruction level. The framework has reasonable runtime overhead and by using a novel compression scheme, we significantly reduce the size of traces. Our framework enables building a wide variety of tools for understanding program behavior. As examples of the applicability of our framework, we present a program analysis and a data locality profiling tool. Our program analysis tool is a time travel debugger that enables a developer to debug in both forward and backward direction over an execution trace with nearly all information available as in a regular debugging session. Our profiling tool has been used to improve data locality and reduce the dynamic working sets of real world applications.

***Categories and Subject Descriptors*** D.2.5 [*Software engineering*]: Testing and debugging

***General Terms*** Algorithms, design, verification

***Keywords*** Code emulation, tracing, callback, code replay, time-travel debugging

## 1. Introduction

The growing complexity of modern software systems makes it increasingly challenging for software designers, implementers, and maintainers to understand system behavior. Program execution traces provide the most detailed description of a program's dynamic behavior. Consequently there has been a lot of interest on gathering and analyzing program execution traces to optimize programs, diagnose failures, and collect software metrics [1, 19, 24]. Traditionally, these techniques have been based on instrumenting a program. The instrumentation can be done either at design time by programmers, build time by a compiler, post-build time by a binary translator [1, 21, 22], or at run time using dynamic binary translation [2, 4, 6, 10, 19].

There exist two major obstacles in program analysis and optimizations when they are based on execution traces. First, running an instrumented binary or executing a program under a runtime entails significant execution overhead. With a flexible instrumentation framework [1, 21, 22] and careful choices of instrumentation points, one can reduce execution overhead. Data collected this way is relevant to a particular task, e.g. program optimization [15, 16] or diagnosis [1]. Second, sizes of execution traces are significant even for a short execution scenario and after applying selected compression algorithms [24]. Sizes of traces can be reduced by limiting output information. However, the applicability of such traces is also limited to a particular task.

In this paper we describe a runtime framework with the following goals:

a) collect a complete trace of a program's user-mode execution,

b) keep the tracing overhead for both space and time low, and

c) re-simulate the traced execution deterministically based on the collected trace with full fidelity down to the instruction level.

Our framework consists of two main components: a run-time engine called Nirvana and a trace recording and retrieving facility called iDNA (Diagnostic infrastructure using Nirvana). Our approach does not require any static instrumentation of the code and makes no assumptions of the analysis task for which the trace will be used. The run-time engine uses a combination of dynamic binary translation and interpretation to emulate the instruction set of a target machine. During emulation, it inserts callbacks to a client application that records information that is sufficient to re-simulate the application's execution at a later time. Recorded information can include attributes of the guest instruction being executed (e.g. the address of the instruction, the address and the read or written value of a memory location if it is a memory-accessing instruction), events like module loads, exceptions, thread creation, etc. It can also include information about where to locate symbolic information, if available. However, Nirvana itself does not require source code or additional symbolic information for code discovery. It can execute dynamically generated code. It also addresses issues of self-modifying code that arise in JIT-compiled and scripting languages. Nirvana supports multi-threaded applications running on multiple processors.



*VEE'06* June 14–16, 2006, Ottawa, Ontario, Canada.

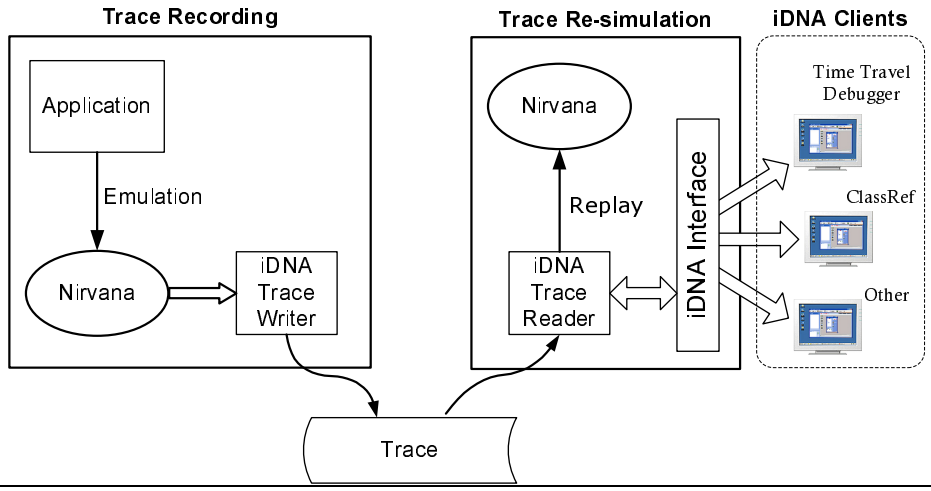


**Figure 1.** Components of our tracing framework.

Our framework is designed for tracing applications executed in user mode rather than the operating system: the claim to full fidelity refers to the behavior observable in user mode only. Since we do not trace kernel mode execution, our system in its current form would be of limited use to analyze operating system bugs. However, any pertinent changes in kernel mode operation that can affect an application's behavior are captured. For example, consider an asynchronous I/O that results in a callback to the application. Although we do not capture the I/O that happens in kernel mode, we do detect the callback in user mode and any subsequent instruction that reads the results of the I/O operation will get logged. Our approach is based entirely on software; it does not require any special hardware or changes to an operating system[1].

The advantages of such an approach are:

- It is not necessary to anticipate a priori all events of interest and have appropriate instrumentation logic for them inserted in the code at trace collection time. Instead, we record a complete program execution trace that can be analyzed multiple times from different perspectives by building different trace analyzers.
- The trace can be gathered on demand by attaching to a running process. There is no performance overhead of the running process when tracing is off.
- The trace can be collected on one machine and re-simulated on another machine or even multiple machines, which allows one to harness multi-processing capabilities of more powerful machines to analyze traces efficiently. It also allows one to perform live analysis by streaming the results to a different machine. Other systems requiring re-simulation of a trace must do it on the same machine (e.g. [14]).
- It reduces the software engineering effort for building and debugging new tools for trace analysis. Since all tools share a common infrastructure for trace gathering and retrieving, tool writers can focus on their analysis engines instead of optimizing and debugging their tracing infrastructures.

The trade-off for this flexibility is a performance overhead both in time and space. Our current performance overhead is approximately a $12 - 17\times$ slowdown of a cpu-intensive user-mode application. This makes it impractical to have it turned on all the time in a production environment. However, the overhead is not prohibitive and in many desktop applications that are not cpu-bound it is practical to run applications in full tracing mode all the time and get a rich repository of traces for dynamic analysis. To demonstrate the practical applicability of this approach, we describe two tools that we have built on our framework:

1) a time traveling debugger, and
2) a data-locality profiling tool that collects and analyzes traces to help optimize data layout for better locality.

The novel contributions of our paper are as follows:

- A compression scheme that enables us to get a complete instruction-level trace with low space overhead and modest runtime overhead. Our scheme yields a compression rate of about 0.5 bits per dynamic instruction instance. It strikes a balance between trace collection and re-simulation speed (affected by compression and decompression) on one side, and compression rate on the other side. While it is possible to get much better absolute compression rates, the runtime overhead for compression and decompression would be quite significant.
- An event-driven run-time instrumentation interface that makes it very easy to build new tools for tracing and analysis and to re-simulate a previously recorded execution. Unlike other systems, our system allows instrumentation to be done either at tracing time or during replay thereby providing a very flexible framework for building offline analysis tools.
- A set of engineering decisions that enables building practical tools on top of our framework, which includes:
  - A code cache management scheme that scales well with multiple threads and processors.
  - A technique to gather traces of multi-threaded applications on multi-processors with very little additional overhead.
  - A practical technique to handle self-modifying code.

The rest of the paper is organized as follows: Section 2 describes the architecture of our system and goes into details of our run-time engine and its interface. Section 3 describes the trace recording, compression, and playback components. Section 4 gives a brief description of two applications we have built on top of this infrastructure. Section 5 presents experimental results to show the time and

[1] Although a minor change in the operating system can make it easier for gaining control of applications (see Section 2)

**Original (guest) code**:

```
mov eax,[ebp+4]
```

**Translated (host) code**:

```
mov esi, nirvContext._ebp
add esi, 4                    ; callee saved register
mov edx, esi                  ; 2nd argument ↦ address of memory accessed
mov ecx, nirvContext          ; 1st argument ↦ pointer to NirvContext
push 4                        ; 3rd argument ↦ number of bytes accessed
call MemRefCallback           ; calling convention assumes ecx and edx hold 1st and 2nd argument
mov eax, [esi]                ; simulate intended memory read
mov nirvContext._eax, eax     ; update NirvContext with new value
```

**Figure 2.** An example of Nirvana's binary translation of a guest instruction.

space overheads of recording and re-simulating traces. Section 6 describes related work.

## 2. Nirvana Runtime Engine

In this section, we describe our runtime engine called Nirvana that is the core component of our tracing and recording framework (Figure 1). Our framework is organized as a 3-layered architecture. The bottom layer is the Nirvana runtime engine that simulates a guest process (i.e. the application being monitored). The middle layers (iDNA Trace Writer and iDNA Trace Reader) use the simulator to record and then re-simulate the guest process. The upper layer contains the task-specific iDNA client, such as a debugger, which takes information in the trace and presents it to the user.

Nirvana simulates a given processor's architecture by breaking the processor's native instructions (called guest or original instructions/code hereafter) into sequences of lower level micro-operations (called host or translated instructions/code hereafter) that are then executed. Breaking the instructions into micro-operations allows Nirvana to insert instrumentation into each stage of the instruction's data path. For example, it may insert instrumentation at the instruction fetch, memory read, and memory write.

Nirvana runs on top of the operating system in user mode. It provides an API that can be used to either launch an application under its control or attach to a running process. We have implemented two mechanisms by which Nirvana can gain control of an application being monitored. The first mechanism relies on a small set of minor changes to the OS that allows Nirvana to register for control on a particular process. These changes are identical to the changes required to give debugger notifications and should be straightforward to support on any operating system. Whenever the OS schedules one of the threads of that process, it dispatches a call to Nirvana and provides the starting address and register context for that thread. The second mechanism relies on a driver that traps all transitions from kernel to user mode and transfers control to Nirvana.

### 2.1 Binary Translation

Upon gaining control of an application, Nirvana emulates the instruction stream by breaking the guest instructions into sequences of simpler operations. Figure 2 illustrates an example of binary translation performed by Nirvana for a single instruction. This example shows the code generated when a client registers for notification on every memory access (see Section 2.4). The original instruction `mov eax,[ebp+4]` reads a value from the memory. During translation Nirvana generates additional code such that it can notify the client about the event. Nirvana keeps on the side the client's execution context (*NirvContext*). In order to execute the client instruction, Nirvana performs the memory arithmetic operation on the client's execution context stored in variable `nirvContext`. Then, it prepares registers for the client's callback function. After the callback function, the instruction is executed and Nirvana updates the client's execution context.

This approach to binary translation in Nirvana differs from other binary translation systems such as [2, 6]:

- The simpler operations used by Nirvana are supported by most architectures. This approach is similar to that used in Valgrind [19] and makes it easy to extend our system to run on other host architectures.
- Provides the ability to monitor code execution at instruction and sub-instruction level granularity with low overhead.
- Provides a framework that works at process level as opposed to the entire system (operating system and all running applications) [14].

Nirvana's goal is to provide a flexible environment and rich set of dynamic instrumentation points such that a full execution trace can be recorded. This includes providing access to each executed instruction as well as each stage of the execution of an instruction, and enabling triggering of instrumentation callbacks by events at the instruction or sub-instruction level. Consequently, running translated code during emulation is much slower than native execution.

Nirvana supports a pure interpretation mode as well as a translation mode with code caches [20]. Code caches are used to store translated code so that if the program reuses an execution path, the binary translator can reuse the translated code. By default, Nirvana uses a code cache. However, under certain cases, described later, Nirvana can switch to the interpretation mode.

### 2.2 Code Cache Management

Three important design issues that affect the code cache performance are:

(*i*) number of code caches,

(*ii*) replacement policy, and

(*iii*) size of code cache.

Ideally, there should be a single code cache that is shared by all threads running on multiple processors. This produces the smallest code working set. However, such a policy creates contention on a multi-processor machine while the code cache is being populated. There are techniques that can reduce the contention [20]. However, they create additional complications and performance overheads in code cache management such as code fragmentation and generation of relocatable code. On the other hand, the simplest approach of having a per-thread code cache does not scale for applications that have tens or hundreds of threads.

| Event name | Description |
|---|---|
| `TranslationEvent` | when a new instruction is translated |
| `SequencingEvent` | start of a sequence point and other special events |
| `InstructionStartEvent` | start of every instruction |
| `MemReadEvent` | memory read |
| `MemWriteEvent` | memory write |
| `MemRefEvent` | memory reference |
| `FlowChangeEvent` | flow control instruction (branch, call, ret) |
| `CallRets` | calls and returns |
| `DllLoadEvent` | load of a new module |

**Table 1.** Sample callback events supported by Nirvana.

Nirvana uses a hybrid approach. The number of code caches is bounded by $P + \delta$ where $P$ is the number of processors and $\delta$ is a small fixed number. Each thread uses a dedicated code cache tied to the processor it is running on. When Nirvana detects a thread voluntarily leaving the code cache (e.g., making a system call), it saves the state of that thread in thread-local storage and returns the code cache to a pool where it is available for reuse by the next thread scheduled on that processor. A thread does not leave its code cache upon a timer interrupt, or while it is executing client callback code, even though the OS may place it in a wait state. In such cases the code cache cannot be reused, and Nirvana will use one of the extra $\delta$ code caches to simulate another thread's execution if there is one available. If none of the extra code caches are available, Nirvana falls back to interpretation on that thread.

We use a default code cache size of 4 MB, which can be overridden by a client. This cache is used to hold both the translated code as well as the mapping tables to map an original instruction to its translated copy in the code cache [20]. The two regions are grown in opposite directions from either end of the code cache and when space gets exhausted, it triggers a code cache flush. While more sophisticated code cache flushing policies can be implemented [13], we did not need to go beyond our simple scheme since empirical data on our target applications show that this policy in combination with other parameters of our code cache management works quite well: the time spent in JIT translation is typically between 2% and 6% of the total tracing time.

### 2.3 Self-Modifying Code

Self-modifying code is typically found in scripting languages and high-level language virtual machines (VM) (e.g. Microsoft's Common Language Infrastructure (CLI) and Java VM's). They use JIT compilation to dynamically compile code in their own code blocks which can be freed and reused. Nirvana uses several techniques to detect and handle self-modifying code. For detection, the techniques used are snooping on specific system calls and/or inserting checks at specific execution points. When self-modifying code is detected, Nirvana handles it by either flushing the code cache and re-translating code or interpreting.

For the correct execution on certain platforms (such as IA64), it is necessary to insert a special OS call to indicate when a code block is modified. Nirvana snoops for these calls and triggers a code cache flush when it detects such a call. The default behavior is not to rely on the presence of such calls. However, it is an optimization that a client can configure Nirvana to use instead of the more expensive detection schemes.

On older implementations of high-level VMs and for applications where it is not known whether there are other instances of self-modifying code, Nirvana monitors the page protection of the code page. If it is a non-writable page, Nirvana takes that to be a strong hint that the code is not modifiable, marks that page as non-writable and translates all code from that page as usual. If it detects that code is from a writable page, Nirvana assumes that the page could be self-modifying. For such code, it inserts a self-modifying check at specific execution points (e.g. instruction boundaries) to make sure that the code has not been modified since it was translated. Nirvana also continually snoops on system calls to check whether the page protection of a code page is changed from non-writable to writable, and triggers a code cache flush if so. Finally, it monitors the set of code cache flushes and if the rate is above a threshold, it sets all pages to be writable and falls back on the slower code that explicitly checks for self-modification. Another alternative is to fall back on interpretation for code pages that are found to be flushed too often. In our experiments we have not encountered such a situation.

### 2.4 Nirvana API

Nirvana's API set is simple and compact and was designed to make it very easy to write applications that can dynamically instrument and observe program behavior. It uses an event-based model that clients can use to register callbacks. There are just two basic functions required for tracing an application:

| | |
|---|---|
| `SetRuntimeOptions()` | This API is used to override any default parameter, like code cache size, interpretation mode, etc. |
| `RegisterEventCallback()` | This is the core function that instructs Nirvana to insert a callback to the client on specific events. |

A sample set of events is presented in Table 1. Note that an event named `SequencingEvent` encompasses all special events on the system which include system calls, exceptions, thread creations, thread terminations, etc. A parameter in the callback indicates which specific `SequencingEvent` triggered the notification.

In addition, we have implemented APIs that facilitate an emulation of a specific instruction stream that is useful e.g. to re-simulate a captured execution trace:

| | |
|---|---|
| `CreateVirtualProcessor()` | Create a virtual processor to emulate a particular ISA |
| `DeleteVirtualProcessor()` | Cleanup and release resources held by the virtual processor |
| `SetVirtualProcessorState()` | Set register state for the virtual processor |
| `GetVirtualProcessorState()` | Query register state for the virtual processor |
| `ExecuteVirtualProcessor()` | Begin emulation of the virtual processor |

Nirvana also provides some convenience utility functions to log information robustly and efficiently that we do not describe in this

paper. Figure 3 shows a sample Nirvana client that can be used to record references to all memory addresses that are read or written to by a guest application. As can be seen it is trivial to write in less than half a page of code a powerful client using Nirvana's API.

## 3. iDNA Tracing and Re-Simulation

In this section we present the tracing component called iDNA. iDNA consists of two main components: iDNA Trace Writer that is used during trace recording and iDNA Trace Reader that is used during re-simulation from a recorded trace. Figure 1 shows the interaction between iDNA components, Nirvana, and the guest application being analyzed. The guest process is simulated by Nirvana, which takes over the user mode execution of the guest process for each thread. The simulator instruments the code stream with callbacks that are defined in the iDNA Trace Writer. The callbacks gather information about the execution that the iDNA Trace Reader needs to re-simulate later. This information is stored in a trace file. Each execution trace contains all the information necessary to reconstruct the entire execution process. This information includes code bytes actually executed and the state of each memory location that is accessed during execution (see /refsec:IDNA:compr). iDNA also records special execution events such as exceptions and module loads. This way, one can re-simulate the trace with full fidelity down to the instruction level.

The iDNA Trace Reader uses Nirvana to re-simulate a program's execution from the information in the trace. As in the traced execution, the simulator uses callbacks into iDNA Trace Reader to request code and data stored in the trace. A trace-analysis client application interacts with the iDNA Interface. The iDNA Interface provides functions to enable efficient transitions between execution points in a trace in both forward and backward directions. It also provides functions for navigation between arbitrary execution points in the trace. Table 2 presents a sample of the iDNA API for navigation through a trace file. This API is based on a standard debugger API enhanced with functionality specific for backward execution. iDNA automatically reconstructs the state of execution for each execution point and provides it on demand to the client. For example, at any point in time a trace-analysis client application can retrieve the register state of a thread or the value of an address in memory.

Although iDNA Trace Writer achieves very good compression during recording, in many cases a complete execution trace of an application from the beginning to end is not needed. For example, when analyzing program failures one usually is interested only in the last portion of the execution history to diagnose the failure. To support such scenarios iDNA uses a circular (or ring) buffer whose size is specified by the user. iDNA records the trace into the ring buffer, and when the buffer is full it goes back to the beginning of the buffer and overwrites the data there. This feature enables the client application to significantly reduce the trace size for many tasks and yet capture enough information for analysis. The default ring buffer size is 16MB, which is sufficient to debug many bugs.

In the remainder of this section we describe further details of recording and re-simulating of execution traces.

### 3.1 Trace Recording for Re-simulation

Conceptually, the log file holds all guest code bytes executed, registers, and data values at every dynamic instance of instructions during the execution. Recording all of that information would not be practical from a time or space standpoint, so iDNA employs multiple compression techniques to drastically reduce the amount of data it needs to record. These techniques take advantage of the fact that we can use Nirvana during re-simulation to regenerate the execution state on each thread.

The iDNA Trace Writer logs all the inputs the Nirvana simulator will need to re-simulate the program. The inputs include the original (guest) code bytes, the register state after a kernel-to-user transition, the register state after certain special instructions whose effects are time or machine specific (e.g. the RTDSC and CPUID instructions on IA-32), and the memory values read by the executing instructions.

With the inputs to the simulator recorded in the log file, the iDNA Trace Reader can feed Nirvana an instruction pointer (IP) and the current register state of a thread at an instance of the instruction. Nirvana then starts executing from that IP in the same way it does during live execution. Because Nirvana is simulating instructions, it automatically keeps the register and IP state current for each instruction that executes.

#### 3.1.1 Data Cache Compression

Employing Nirvana during re-simulation to re-generate memory and register data drastically reduces the amount of data the iDNA Trace Writer needs to log to the trace file. iDNA Trace Writer still needs to log all the values read/written by the instruction stream from/to memory. Logging every one of those reads creates excessively large logging overhead. In order to reduce the overhead, we have designed a compression algorithm that limits the amount of data we need to store in the trace file. The key insight for reducing the data size is to recognize that *not all memory values that are read need to be recorded; we only need to record those memory reads that cannot be predicted.*

One of the components of the iDNA Trace Writer is a tag-less direct mapped cache for each guest process thread. The cache is designed to hold the last accessed value for memory accesses for the thread. The cache is indexed by the accessed address. The cache buffer is initialized to all zeros. Every time an instruction reads a value from memory, the value is compared to its mapped position in the cache. The value is logged in the trace file only if the cached value is different from the actual value read from memory. Reading a different value from a memory location compared to previously written value to the same location can happen due to a change to the memory location during kernel mode execution, DMA, or by a thread running on a different processor. Otherwise, a counter is incremented that keeps track of how many reads are correctly predicted. When a value does not match the cached value, the cache is also updated. The read prediction rate is enhanced even further by taking all data values written by the instruction stream and writing them directly to the cache. The iDNA Trace Reader uses the same type of cache during re-simulation. The only difference is that the iDNA Trace Reader uses the trace file instead of the monitored application to update the cache when the instruction stream read corresponds to a logged read. The reader re-directs the simulator's instruction and data fetch operations to point to the stored code bytes and data cache buffer.

The example in Figure 4 shows how our data cache works to eliminate the need to store data values in the trace file[2]. The iDNA Trace Writer will receive notification when the address in memory representing both *i* and *j* are written. Those values are always stored directly into the cache. The reads of *i* and *j* in the loop are always predicted correctly, since their cache values are updated by previous writes. However, after the call to *system_call()*, when the value of *i* is read it is not the predicted value of 46 (presumably because it changed during the system call). So in this example, the iDNA Trace Writer will have one record for *i* with its initial value of 1 and a successful prediction count of 11 (since *i* was predicted correctly 11 times) followed by another record for *i* with a value of

[2] We assume in this example that the values of i and j are not in registers, but read directly from memory.

```
void __fastcall MemCallback(NirvContext *pcntx, void *pAddr, void* nBytes)
{
  X86REGS *pregs = (X86REGS *)pcntx->pvCpuRegs;
  Log(pregs->InstructionPointer(), pAddr, nBytes);
}

// Initialize is a special routine that is called once Nirvana launches
// or attaches to a process.

extern ''C'' __declspec(dllexport) bool __cdecl Initialize()
{
  if (InitializeNirvanaClient() != FALSE)
  {
      RegisterEventCallback(MemRefEvent, MemCallback);
  }
}
```

**Figure 3.** Sample Nirvana client code to get a trace of all memory references made by an application

| API name | Description |
|---|---|
| `ExecuteForwared()` | Execute forward the specified number of steps or until the next breakpoint. |
| `ExecuteBackwards()` | Execute backward the specified number of steps or until the previous breakpoint. |
| `JumpToPosition()` | Set the iDNA Trace Reader execution state to the specified position. |
| `SetPositionBreakPoint()` | Set a breakpoint to a specific position in the trace. |
| `SetExecutionBreakPoint()` | Set a breakpoint to a specific execution address. |
| `SetMemoryBreakPoint()` | Set a breakpoint to an access of a specific memory location. |
| `SetEventBreakPoint()` | Set a breakpoint on a particular event (e.g. module load, exception, etc.). |
| `ClearBreakPoint()` | Remove one or more breakpoints. |
| `FindPrevWrite()` | Find the previous write to a specified memory location relative to the current position. |
| `FindNextWrite()` | Find the next write to a specified memory location relative to the current position. |
| `FindPrevRead()` | Find the previous read from a specified memory location relative to the current position. |
| `FindNextRead()` | Find the next read from a specified memory location relative to the current position. |

**Table 2.** iDNA API functions that facilitate navigation through a trace.

```
i = 1;
for (j = 0; j < 10; j++)
{
  i = i + j;
}
k = i; // value read is 46
system_call();
k = i; // value read is 0
```

**Figure 4.** An example that illustrates data compression.

0. It will not record any further values for either *i* or *j* as long as their cached values match the observed values. During re-simulation, the iDNA Trace Reader reads the initial value for each memory location from the trace log and decreases the prediction count each time the memory location is read. When the count goes to 0, it gets the new value for that location from the next record in the trace file.

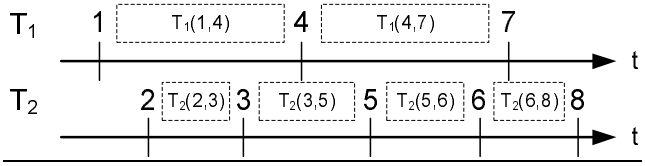


**Figure 5.** Sequencing between two threads.

### 3.2 Scheduling

iDNA records information provided by Nirvana into a file on disk. In order to avoid introducing contention between different threads of the process and to reduce I/O, iDNA buffers the data for each thread independently in different streams in shared memory. The buffers are flushed to disk asynchronously either when the buffer is full or the thread or process terminates. Within each stream we record the data needed to simulate the program execution from the trace file at a later time. The data within the file represents the sequence of instructions executed while recording, and each executed instruction is a unique point in time. Because each thread's execution is independently recorded, at certain events within the program execution the iDNA Trace Writer creates ordering points or *key frames*. Key frames enable the reader to know how sequences of instructions relate to each other across multiple threads.

#### 3.2.1 Instruction Sequencing

*Key frame* ordering is generated from an incrementing counter that is global to the guest process. A key frame is logged to a thread's stream whenever:

- the thread enters or leaves user mode execution,
- a synchronization operation is executed, or
- a memory barrier operation is executed.

A synchronization operation is any instruction or combination of instructions that is used to serialize execution. For example on

x86 architectures, instructions that have a `lock` prefix and the `xchg` instruction (which is implicitly atomic) are synchronization operations.

Between two key frames is an instruction sequence. Within a single thread all instructions in a sequence are ordered by when they are executed. Between threads they are ordered by the counter stored in the key frame at the start and end of each instruction sequence. We generate key frames, as opposed to using a counter to individually order every instruction, to save on both space in our log file and logging time overhead.

In the example shown in Figure 5, thread $T_1$ has three key frames: 1, 4, and 7. The first instruction sequence occurs between frame 1 and 4. We denote this sequence as $T_1(1,4)$. Using our method we know that all instructions in sequence $T_1(1,4)$ are executed before $T_2(5,6)$ and $T_2(6,8)$ on the second thread. This is because the start of those sequences begin after sequence $T_1(1,4)$ ends. However, because execution on $T_1(1,4)$ overlaps $T_2(2,3)$ and $T_2(3,5)$ we cannot order the instructions between threads on those sequences. This is one reason why generating a key frame on every synchronization operation is important. It ensures that thread sequencing events in the code that order statements are preserved during re-simulation.

In addition to these key frames, the iDNA Trace Writer also generates checkpoint frames (also called key frames hereafter) at regular intervals (currently after 5 MB increment of the logged data size) where it records the thread context (values of all registers, flags, floating point state) and flushes the data cache. These checkpoint frames are not necessary for correct re-simulation, but are an optimization that allows re-simulation to begin from random places in the trace instead of having to always go to the beginning of the trace.

With this method we can get a total ordering of instructions on a single processor system by tracking context-swaps between threads. On multi-processor systems, no total ordering exists because instructions on different processors run concurrently. However we guarantee that all synchronization and memory barrier instructions are totally ordered. This is sufficient to analyze an execution trace for certain kinds of data races, e.g. by detecting missing synchronization instructions when accessing a shared memory location.

### 3.3 Sequence-Based Scheduling

The execution trace conceptually holds all the guest instructions executed in each guest process thread. The ordering of any particular instruction during re-simulation is determined by what sequence it is in. On multi-processor systems, it is not possible to determine precise time ordering of the instructions based on their trace positions. Thus, the iDNA Trace reader implements a scheduler that determines what instruction sequence is next to execute across all the threads. We call this sequence-based scheduling.

As shown in Figure 6, the scheduler schedules all instructions within a sequence before it moves onto the next sequence. The ordering is based on the sequencing counter number at the key frame that starts the sequence. The resulting ordering of instructions is likely to be different than the precise time ordering of instructions during execution. However, the ordering of key events (kernel-user transitions, instances of synchronization instructions, and instances of memory barrier instructions) is preserved.

## 4. Applications

An efficient instruction level trace opens a whole new field of possible applications. We have deployed Nirvana internally within Microsoft and it is being used by other researchers and product groups to build applications in security [9] and performance analysis. We have built several applications that have also been deployed internally within Microsoft. We are going to describe two of them - time travel debugging and data locality profiling. A third major application is focused on finding bugs by analyzing traces, but a description of that is beyond the scope of this paper. All of these applications are in routine use by several product groups within Microsoft.

### 4.1 Time Travel Debugging

Time travel debugging is the ability to go arbitrarily forward and backward in time in a debugger to debug a program. Time travel debugging brings a whole new dimension to diagnosing and understanding a program since in addition to the typical "forward" debugging experience, a programmer can now travel backwards through the execution of their program to interesting points.

We have integrated this technology into a traditional debugger. After loading a trace into the debugger, the user can travel to the last exception that occurred in the trace, and then examine the memory and register state at that instant in time. If the exception was caused while de-referencing an invalid memory address, he or she can set breakpoints on the memory location containing the invalid address, and travel backwards to the point in time where that memory was last written to before the exception. On the iDNA level of abstraction, we go back to the first key frame before the breakpoint, recreate the simulation state and re-simulate forward to the breakpoint. From the user standpoint, it appears as if the simulation is going seamlessly in the backward direction. Since we have checkpoint frames at regular intervals, going backwards is very efficient: in a debugger the response time for stepping back is indistinguishable from the response time for going forward during a live debugging. This can make diagnosing difficult bug almost trivial. Note that the client application does not need to handle issues related to positioning in the trace. All positioning is handled by the iDNA layer. Table 2 presents the set of iDNA APIs that facilitates navigation through a trace file.

The key functionalities we added to the debugger include:

- Querying the current "time" (i.e. the current position in the trace)
- Stepping backwards
- Backwards code breakpoints
- Backwards data (read and/or write) breakpoints
- Running backwards (until a breakpoint)
- Jumping to a specific "time"

The first implementation of time travel debugging took about 3-4 man-months of effort which includes the learning time to become familiar with the debugger code base. Most of the effort in integration was to modify the debugger interface that queries for memory values (which could contain code or data) at a given point in time. The trace reader component provides the values of these code and data values by time traveling back in time to the last place where that memory value was updated. An additional 4-6 man-months of effort was used to implement various optimizations to get good response times for common debugging operations and make it robust enough for deployment.

The integration of tracing with the debugger should have a dramatic impact on developer productivity when developers are debugging non-deterministic bugs and/or bugs that are hard to reproduce. We have implemented several optimizations and indexes that make it efficient to do common tasks during debugging like examining call stacks, local variables, stack values, and memory locations that are temporally close to the current position. With these optimizations, the response time for most of these tasks is almost the same as in a live debugger session.

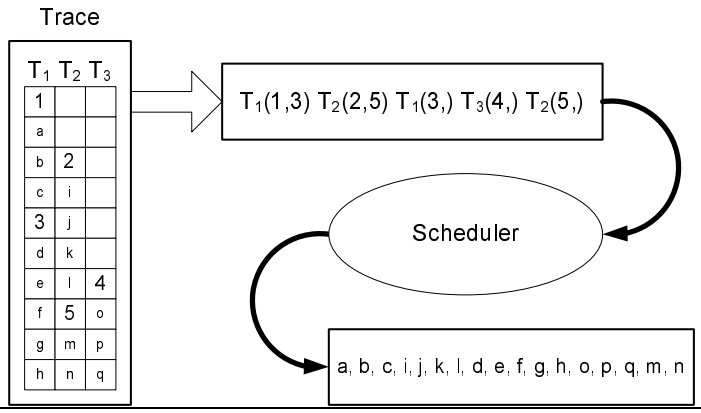


**Figure 6.** Sequence based scheduling.

### 4.2 Data Locality Profiling

The second application we developed is a tool called ClassRef, which has been used to optimize how applications use dynamic memory in a garbage collected environment. ClassRef takes advantage of the fact that our trace contains all of the information about what virtual memory addresses are referenced by an application. Using that information, our tool displays reference data for objects in the garbage collected heap. For a specified time period it shows which fields in an application's classes are referenced and which instances of those classes are referenced. In addition it tracks down potential memory leaks and maps that data back to the source code.

ClassRef has been used to find opportunities for improving data locality. Improving data locality may involve implementing some of the following solutions:

- Class re-factoring
  - Splitting a class
  - Removing fields
- Class layout improvements
- Nulling objects that are no longer referenced
- Changing allocation patterns to keep heavily referenced objects together

For example, ClassRef may show that a field in a heavily used base class is only used in one of its derived classes. In that case, moving the field from the base to the derived class may result in significant memory savings. ClassRef may also show a spike in the number of uncollected and unreferenced objects in an application while running in steady state. This may indicate that the application is holding references to objects, preventing the garbage collector from doing its job. ClassRef has been used to help obtain significant savings of dynamic working sets in real applications (unpublished report from an internal product that is currently under development).

## 5. Experimental Results

Nirvana currently supports the x86 family of CPUs and is currently being extended to support x64. Our tracing framework has been used to gather traces on hundreds of program scenarios and is in routine use by users to trace many different kinds of applications. In this section, we present the effectiveness of our framework on a set of benchmarks that consists of a SPEC2000 application, two desktop applications, Internet browser, a binary disassembly application, and a C# application. We have chosen this set of benchmarks in such a way that it contains representatives of applications that are computation intensive (Gzip and SatSolver), user interface (UI) intensive (Spreadsheet and Presentation), and I/O intensive (Internet browser and DumpAsm[3]). SatSolver is an example of an application with self-modifying code that our framework handles successfully. We conducted all the experiments on an Intel P4 processor on 2.2 GHz with 1GB RAM.

In Table 3, we present tracing overhead of our framework. We compare execution times of benchmark applications running in native mode with execution times under the tracing environment of our framework. The overhead ranges from 5.66 to 17 times the native execution times. Our framework exhibits the smallest overhead with UI intensive applications. This low overhead is the consequence of high volume of I/O operations, which means that the processor is not utilized close to 100 %. The time to re-simulate a traced application is currently about 3-4 times the cost of tracing. We expect that this can be optimized to be on par with tracing time. The cost during re-simulation is usually dominated by the analysis task, so we have focused more on optimizing commonly used operations during analysis instead of the raw re-simulation time.

The compression rates of trace files yielded by our framework are illustrated in Figure 7. We define the compression rate as the ratio of the trace file size in bits and the number of executed instructions for each benchmark application. The compression rates range from 0.08 bits per instruction for Gzip to 1.1 for SatSolver. Average compression ratio for the benchmark set is 0.51 bits per instruction. Not surprisingly, the worst compression rate is obtained for the C# application because concurrent garbage collection interferes with our data prediction mechanisms. The best compression rates are encountered for applications that perform long and repetitive local computation on data, which enables our prediction mechanism to be highly accurate.

## 6. Related Work

There exist a number of run-time systems that monitor program execution. DELI [10] is based on earlier approaches (Dynamo [2] and DynamoRIO [4]). These systems were originally designed for run-time optimizations. They allow client applications to control the execution flow of an application. However, these approaches do not provide infrastructure for collecting fine-grain program execution information (e.g. value of an accessed memory location). They leave it up to a client application to handle many engineering details (e.g. the management of self-modifying code and multi-threaded

[3] DumpAsm is an in-house application that disassembles binaries and prints out their assembly code.

| Application Name | Instructions (millions) | Compression rate [bits/instruction] | Trace size [MB] | Native time [s] | Tracing time [s] | Tracing overhead |
|---|---|---|---|---|---|---|
| Gzip | 24 097 | 0.08 | 245 | 11.7 | 187 | 15.98 |
| Spreadsheet | 1 781 | 0.44 | 99 | 18.2 | 105 | 5.76 |
| Presentation | 7 392 | 0.57 | 528 | 43.6 | 247 | 5.66 |
| Internet browser | 116 | 0.36 | 5.15 | 0.499 | 6.94 | 13.90 |
| DumpAsm | 2 408 | 0.50 | 152 | 2.74 | 46.6 | 17.01 |
| SatSolver | 9 431 | 1.10 | 1 300 | 9.78 | 127 | 12.98 |
| Average tracing overhead | | | | | | 11.89 |

**Table 3.** The comparison of native execution time with execution time while tracing. The last column represents the tracing overhead as a ratio of tracing time and native time.

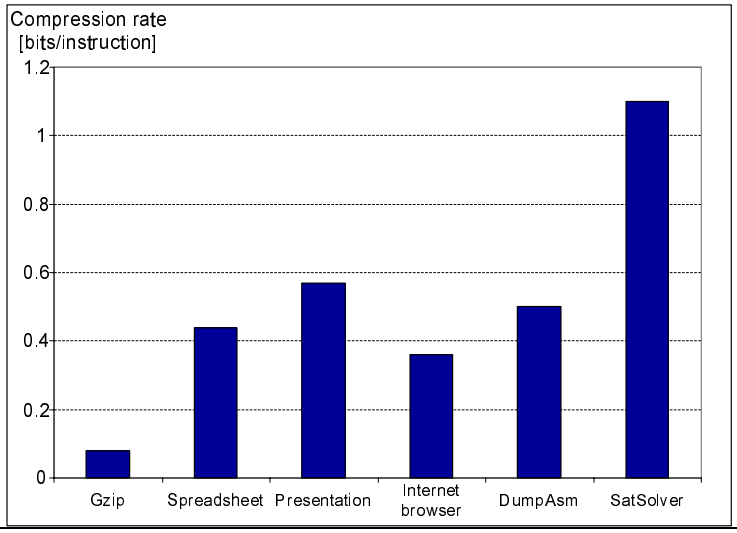


**Figure 7.** Compression rate as the ratio of the trace file size in the number of bits and the number of executed instructions. The number above each bar represents the trace file size.

issues). Shade [8] was one of the earliest work that shares many of the same goals as our work. The JIT translation technique used in Nirvana is quite similar to that used in Shade and with similar performance characteristics. Shade does not provide re-simulation capabilities nor does it deal with tracing deterministically on multi-processors. Another system that is similar to Nirvana is Valgrind [19] which breaks instructions into RISC-like micro-operations and provides instruction level binary instrumentation. However, it lacks a general tracing and re-simulation capability, and handles self-modifying code and multi-threaded applications in an ad hoc manner.

Pin [17] is a newer dynamic binary translator that provides dynamic instrumentation like Nirvana using an API that was inspired by ATOM [21]. It performs several optimizations during translation and it yields tracing overhead that is lower than Nirvana's. However, Pin and its performance is not fully comparable to Nirvana since it is not clear how effective some of the optimizations would be for heavier instrumentation at the sub-instruction level (e.g. every memory read and write) and for multi-threaded applications.

A number of research efforts have been directed towards collection of program execution traces for their later analysis. The main focus of these approaches is handling large trace sizes by compressing them on the fly so that they can be efficiently stored and retrieved. Zhang and Gupta [24] show some of the latest advances in collecting complete execution traces by statically instrumenting a binary. Their approach uses an efficient compression mechanism which allows the reconstruction of the entire program execution. Once the reconstruction is done, all the relevant information about the instrumented location are readily available for analysis. It requires a sophisticated static analysis for both the instrumentation and reconstruction of the trace, and it is not clear how multi-threaded issues and self-modifying code can be handled in their framework. TraceBack [1] is another static instrumentation-based tool that collects and reconstructs control flow information of multi-threaded applications for fault detection.

There are various checkpointing schemes that take a snapshot of the full system state from where execution can restart deterministically [3, 11, 12]. TTVM [14] and FDR [23] both use checkpointing to retrieve system state and record all inputs into the system (I/O, interrupts, DMA transfers) to do deterministic re-simulation. Our approach does not require system state checkpointing or the tracking of system inputs since we only trace user mode execution. The most closely related work to ours is BugNet [18] that also targets debugging user mode execution and also tracks reads from memory. BugNet proposes modifications to the hardware by using a first-load bit to record when an L1 or L2 cache block is replaced. Whenever there is a load of a memory location whose first-load bit is set, the logging of that read is skipped. Our caching scheme is purely software based, and allows us to use a thread-specific cache for each process, instead of using a per-processor cache that is shared across different processes. In BugNet, the first load bits are not tracked across interrupts. They are reset instead

and, in effect, create a new checkpoint interval on every interrupt. The corresponding operation in our scheme is to flush the prediction cache. However, there is no need to flush the prediction cache on interrupts since we do not need to know when the cache is invalidated. Instead, we check on each read whether the value read is the one predicted and if so, logging can be skipped.

There has also been considerable work on trace compression and specifically value prediction-based compression (VPC) [5] that is relevant to our work. VPC algorithms can achieve better compression rate than our scheme, but at a higher performance overhead for compression and decompression.

Time traveling debugging that is based on virtual machine re-simulation has been done by others. ReVirt is one of the latest systems that shows how system level virtual machines can be used to do time traveling debugging of operating systems [14]. Data-locality profiling and optimizations based on profile traces have been reported in others (e.g. [7]), but these approaches are based on static instrumentation of code.

## 7. Conclusion

We have described a framework that can be used to collect detailed traces of a program's user-mode execution and re-simulate those traces with full fidelity down to the instruction level. The framework is based on dynamic binary translation and does not require any static instrumentation of the program. The traces produced are task-independent and self-contained, making it easy to transmit them to other machine for re-simulation and analysis. The framework uses a layered architecture and exposes an event-based API that makes it easy to write tracing and trace-analysis applications.

Our tracing overheads, both time and space, are quite reasonable as demonstrated by empirical data on a variety of applications. We have described engineering decisions that allow our approach to scale well with multiple threads and processors, and self-modifying code. We believe that these efficiencies open up many practical application areas for understanding program behavior from different perspectives. We have described two real analysis applications that we have built on top of our framework and are working on several other applications.

## 8. Acknowledgements

We are grateful to many of our colleagues at Microsoft including Todd Proebsting, Hoi Vo, Kieu Nguyen, Jordan Tigani, Zhenghao Wang, Drew Bliss, and Ken Pierce for their valuable contributions on our framework and tools, and to the anonymous reviewers for their constructive comments on earlier drafts of this paper.

## References

[1] Andrew Ayers, Richard Schooler, Chris Metcalf, Anant Agarwal, Junghwan Rhee, and Emmett Witchel, *TraceBack: first fault diagnosis by reconstruction of distributed control flow*, Programming Language Design and Implementation (PLDI '05) (2005), 201–12.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia, *Dynamo: a transparent dynamic optimization system*, Programming Language Design and Implementation (PLDI '00) (2000), 1–12.

[3] Thomas C. Bressoud and Fred B. Schneider, *Hypervisor based fault-tolerance*, Symposium on Operating Systems Principles (SOSP '95) (1995), 1–11.

[4] Derek Bruening, Timothy Garnet, and Saman Amarasinghe, *An infrastructure for adaptive dynamic optimization*, International Symposium on Code Generation and Optimization (CGO '03) (2003), 265–75.

[5] Martin Burtscher, Ilya Ganusov, Sandra J. Jackson, Jian Ke, Paruj Ratanaworabhan, and Nana B. Sam, *The vpc trace compression algorithms*, IEEE Transactions on Computers **54** (2005), no. 11, 1329–44.

[6] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David Gillies, *Mojo: a dynamic optimization system*, ACM Workshop on Feedback-Directed and Dynamic Optimization (2000), 81–90.

[7] Trishul Chilimbi and James R. Larus, *Cache-conscious structure layout*, Programming Languages Design and Implementation (PLDI '99) (1999), 13–24.

[8] Bob Cmelik and David Keppel, *Shade: a fast instruction-set simulator for execution profiling*, ACM SIGMETRICS (1994), 128–37.

[9] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham, *Vigilante: End-to-end containment of internet worms*, ACM Symposium on Operating Systems Principles (SOSP '05) (2005), 133–147.

[10] Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, and Joseph A. Fisher, *DELI: a new run-time control point*, ACM/IEEE International Symposium on Microarchitecture (MICRO '02) (2002), 257–68.

[11] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen, *Revirt: enabling intrusion analysis through virtual machine logging and replay*, Symposium on Operating Systems Design and Implementation (OSDI '02) (2002), 211–224.

[12] E.N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B.Johnson, *A survey of rollback recovery protocols in message-passing systems*, ACM Computing Survey **34** (2002), no. 3, 375–408.

[13] Kim Hazelwood and James E. Smith, *Exploring code cache eviction granularities in dynamic optimization systems*, International Symposium on Code Generation and Optimization (CGO '04) (2004), 89.

[14] Samuel T. King, George W. Dunlap, and Peter M. Chen, *Debugging operating systems with time-traveling virtual machines*, USENIX Annual Technical Conference (2005), 1–15.

[15] James R. Larus, *Whole program paths*, Programming Language Design and Implementation (PLDI '99) (1999), 259–69.

[16] James R. Larus and Eric Schnarr, *EEL: machine-independent executable editing*, Programming Language Design and Implementation (PLDI '95) (1995), 291–300.

[17] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood, *Pin: building customized program analysis tools with dynamic instrumentation*, Programming Language Design and Implementation (PLDI '05) (2005), 190 – 200.

[18] Satish Narayanaswamy, Gilles Pokam, and Brad Calder, *Bugnet: Continuously recording program execution fro deterministic replay debugging*, International Symposium on Computer Architecture (ISCA '05) (2005), 284–95.

[19] Nicholas Nethercote and Julian Seward, *Valgrind: a program supervision framework*, Electronic Notes in Theoretical Computer Science **89** (2003), no. 2.

[20] James E. Smith and Ravi Nair, *Virtual machines*, first ed., Morgan Kaufman, San Francisco, Ca., 2005.

[21] Amitabh Srivastava and Alan Eustace, *ATOM: a system for building customized program analysis tools*, Programming Language Design and Implementation (1994), 196–205.

[22] Amitabh Srivastava, Hoi Vo, and Andrew Edwards, *Vulcan: binary transformation in distributed environment*, Tech. Report MSR-TR-2001-50, Micorosft Research, April 2001.

[23] Min Xu, Rastislav Bodik, and Mark D. Hill, *A 'flight data recorder' for enabling full-system multiprocessor deterministic replay*, International Symposium on Computer Architecture (ISCA '03) (2003), 122–35.

[24] Xiangyu Zhang and Rajiv Gupta, *Whole execution traces and their applications*, ACM Transactions on Architecture and Code Optimization **2** (2005), no. 3, 301–334.