Dynamic Binary Instrumentation

The term instrumentation refers to an ability to monitor or measure the level of a product's performance and to diagnose errors. In programming, this means the ability of an application to incorporate:

- Code tracing - Receiving informative messages about the execution of an application at run time.
- Debugging - Tracking down and fixing programming errors in an application under development.
- Performance counters - Components that allow you to track the performance of your application.
- Event logs - Components that allow you receive and track major events in the execution of your application.

1 Instrumentation levels

Instrumentation can be done on a program on different levels:

1.1 Source-Level instrumentation

Instrumentation code is added to the program source code either automatically or manually. In other words the program source code is changed and this requires that the source code for the program should be available otherwise instrumentation is not possible.

1.2 Binary-level instrumentation

Binary instrumentation consists of adding trace points to an already compiled application. This may be done statically, before the application is started, or dynamically, while the application is running. The challenge is to add the new instructions, for instance trace points, while not changing the rest of the program (Binary Instrumentation). There are two types of binary instrumentation:

1.2.1 Static binary instrumentation

In this approach all instrumentation codes are added to the compiled application and a new application which includes the instrumentation code is created. Examples of static binary instrumentation frameworks/tools include ATOM, EEL, Etch and Morph. Static instrumentation has many limitations compared to dynamic instrumentation. The most serious one is that it is possible to mix code and data in an executable and a static tool may not have enough information to distinguish the two. Dynamic tools can rely on execution to discover all the code at run time. Other difficult problems for static systems are indirect branches, shared libraries, and dynamically-generated code. [2]

1.2.2 Dynamic binary instrumentation

Dynamic binary instrumentation techniques are used to execute additional instructions at certain locations in a program. Since instrumentation is added dynamically, a tracing overhead is incurred only when a program is dynamically instrumented to produce a trace. An advantage of this approach is that already running programs can be instrumented since it is possible to add instrumentation codes on-the-fly.
Examples of dynamic binary instrumentation frameworks are: Pin, Dyninst, DynamoRIO.

2 DBI framework properties

2.1 DBI mode

Dynamic binary instrumentation can be done in JIT mode or probe mode. In JIT mode the instrumentation framework acts like a “Just-In-Time” (JIT) compiler. The input to this

compiler is not byte code, however, but a regular executable. The framework fetches small sections of the code and the tool written by the framework receives the stream of instructions, inspects this stream using the framework API and inserts instrumentation code where ever necessary. Then the framework re-compiles this section of code, creating a new code fragment which includes the instrumentation code as well.
In probe mode, instrumentation codes are executed at designated points of the original code using “code trampolines”. In other words, the tool provides callback functions to the framework and this functions get called once the specified events occurs. This means the tool does not have the opportunity of inspecting all the instructions but it should tell the framework at which point it wants to insert instrumentation and the framework will callback the instrumentation code when such points are encountered, for example every time a memory read occurs or a certain instruction is executed or a certain function is called.

2.2 Granularity

Different frameworks allow instrumentation code to act on different levels of granularity. The DBI tool developer might use these different levels to ease the task of development. For example function level, instruction level or event level.

2.3 Other properties

In the following sections different DBI frameworks will be explained. Other important properties of a DBI framework includes the hardware architectures supported by the framework and the ability of instrumenting multi-threaded applications. Another important property is the ability of accessing the symbol table of the program (if available).

In the following sections different DBI frameworks will be explained.

3 Pin

Pin is a dynamic binary instrumentation framework that enables the creation of dynamic program analysis tools. It supports Linux and Windows executables for IA-32, Intel(R) 64, and IA-64 architectures. Previous versions of Pin supported ARM but it was discontinued. Some tools built with Pin are Intel Parallel Inspector, Intel Parallel Amplifier and Intel Parallel Advisor. [1]
Pin allows a tool to insert arbitrary code (written in C or C++) in arbitrary places in the executable. The code is added dynamically while the executable is running. This also makes it possible to attach Pin to an already running process.[1]
Instrumentation tools written using Pin API are called Pintools.

3.1 How it works

At the highest level, Pin consists of a virtual machine (VM), a code cache, and an instrumentation API invoked by Pintools. The VM consists of a just-in-time compiler (JIT), an emulator, and a dispatcher. After Pin gains control of the application, the VM coordinates its components to execute the application. The JIT compiles and instruments application code, which is then launched by the dispatcher. The compiled code is stored in the code cache. Entering/leaving the VM from/to the code cache involves saving and restoring the application register state. The emulator interprets instructions that cannot be executed directly. It is used for system calls which require special handling from the VM. Since Pin sits above the operating system, it can only capture user-level code. [2]
As Figure 1 shows, there are three binary programs present when an instrumented program is running: the application, Pin, and the Pintool. Pin is the engine that JITs and instruments the application. The Pintool contains the instrumentation and analysis routines and is linked with a library that allows it to communicate with Pin. While they share the same address space, they do not share any libraries to avoid unwanted interaction between Pin, the Pintool, and the application. [2]
The injector loads Pin into the address space of an application. Injection uses the Unix Ptrace API to

obtain control of an application and capture the processor context. It loads the Pin binary into the application address space and starts it running. After initializing itself, Pin loads the Pintool into the address space and starts it running. The Pintool initializes itself and then requests that Pin start the application. Pin creates the initial context and starts JITing the application at the entry point (or at the current PC in the case of attach).

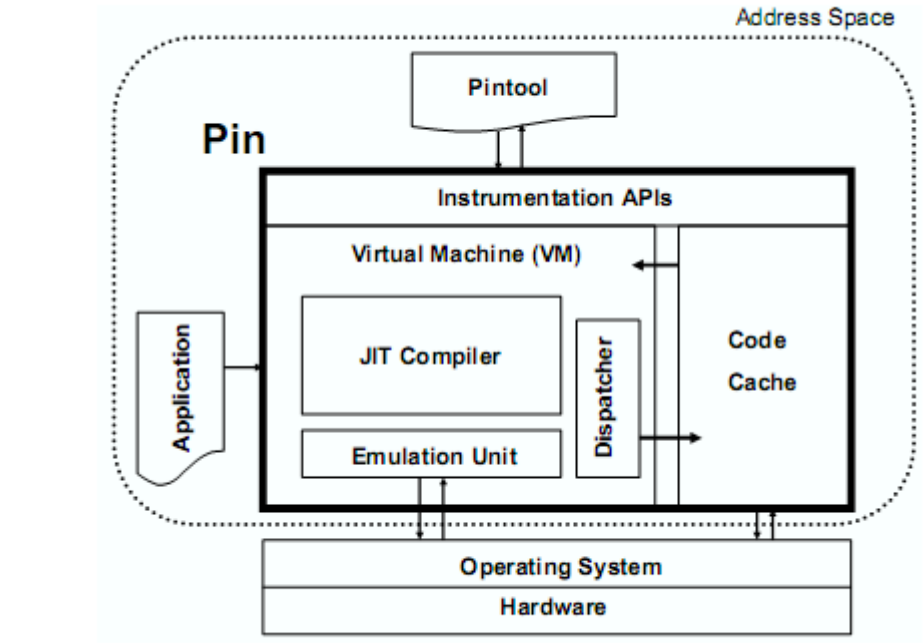


Figure 1: Pin's software architecture

In JIT mode, Pin fetches small sections of the code, recompiles this code and keeps it in a software-based code cache. Only code residing in the code cache is executed, the original code is never executed. An application is (re)compiled one trace at a time. A trace is a straight-line sequence of instructions which terminates at one of the conditions:

- An unconditional control transfer (branch, call, or return)
- A pre-defined number of conditional control transfers
- A pre-defined number of instructions have been fetched in the trace.

The generated code sequence is almost identical to the original one, but Pin ensures that it regains control when a branch exits the sequence. After regaining control, Pin generates more code for the branch target and continues execution. Every time the JIT fetches some code, the Pintool has the opportunity to instrument it before it is translated for execution. The translated code and its instrumentation is saved in the code cache for future execution of the same sequence of instructions to improve performance. [2]
A more detailed description of how Pin works can be found at [3].

3.2 Features

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

- **Instrumenting Multi-threaded Applications:** Instrumenting a multi-threaded program requires that the tool be thread safe - access to global storage must be coordinated with other threads. Pin tries to provide a conventional C++ program environment for tools, but it is not possible to

use the standard library interfaces to manage threads in a Pintool. For example, Linux tools cannot use the pthreads library and Windows tools should not use the Win32 API's to manage threads. Instead, Pin provides its own locking and thread management API's, which the Pintool should use. Pintools do not need to add explicit locking to instrumentation routines because Pin calls these routines while holding an internal lock called the VM lock. However, Pin does execute analysis and replacement functions in parallel, so Pintools may need to add locking to these routines if they access global data.

- **Trace linking:** To improve performance, Pin attempts to branch directly from a trace exit to the target trace, bypassing the stub and VM. This process is called trace linking. Linking a direct control transfer is straightforward as it has a unique target. Pin simply patches the branch at the end of one trace to jump to the target trace (figure 2). However, an indirect control transfer (a jump, call, or return) has multiple possible targets and therefore needs some sort of target-prediction mechanism. Detailed description of trace linking techniques for indirect control transfer can be found at [2].

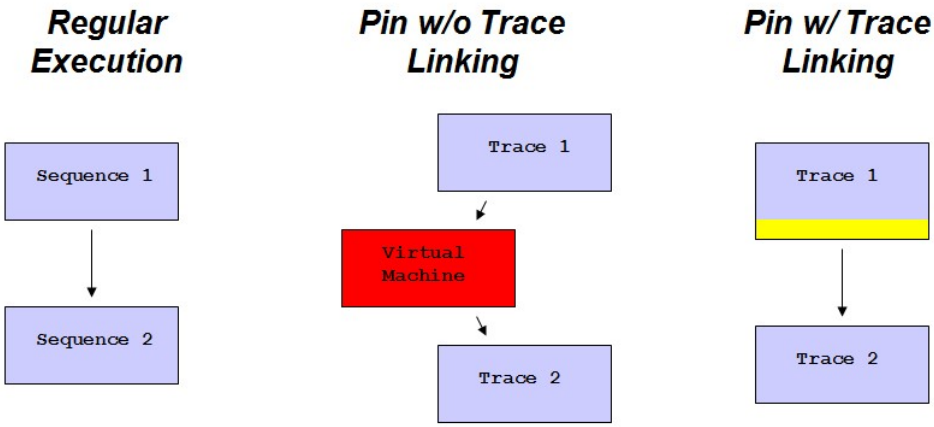


Figure 2: Trace linking in Pin

Pin also provides a limited ability to alter the program behavior by allowing an analysis routine to overwrite application registers and application memory.

3.3 API

Pin provides a rich API that abstracts away the underlying instruction set idiosyncrasies, making it possible to write portable instrumentation tools. The API also allows access to architecture-specific information. The Pin API makes it possible to observe all the architectural state of a process, such as the contents of registers, memory, and control flow. [2]

Pin allows instrumentation code to perform on the target application on four different granularities or modes:

- **Instruction instrumentation:** lets the tool inspect and instrument an executable a single instruction at a time. This is essentially identical to trace instrumentation where the Pintool writer has been freed from the responsibility of iterating over the instructions inside a trace. Instruction instrumentation utilizes the *INS_AddInstrumentFunction* API call.

- **Trace instrumentation:** is the previously explained mode where each time a sequence of code is fetched and recompiled. Trace instrumentation utilizes the *TRACE_AddInstrumentFunction* API call.

- **Image instrumentation:** the Pintool inspect and instrument an entire image, IMG, when it is first loaded. A Pintool can walk the sections, SEC, of the image, the routines, RTN, of a section,

and the instructions, INS of a routine. Instrumentation can be inserted so that it is executed before or after a routine is executed, or before or after an instruction is executed. Image instrumentation utilizes the *IMG_AddInstrumentFunction* API call.

- **Routine instrumentation:** lets the Pintool inspect and instrument an entire routine before the first time it is called. A Pintool can walk the instructions of a routine. Routine instrumentation utilizes the *RTN_AddInstrumentFunction* API call.

The last two modes are implemented by "caching" instrumentation requests and hence incur a space overhead. [1]
The Pintool can also register notification callback routines for events such as thread/process creation or forking, application start/exit and signal interception.

3.4 Using Pin in probe mode

Probe mode is a method of using Pin to insert probes at the start of specified routines. A probe is a jump instruction that is placed at the start of the specified routine. The probe redirects the flow of control to the replacement function. In probe mode, the application and the replacement routine are run natively. This improves performance, but it puts more responsibility on the tool writer. Probes can only be placed on RTN boundaries. Many of the PIN APIs that are available in JIT mode are not applicable in Probe mode. In particular, the Pin thread APIs are not supported in Probe mode, because Pin has no information about the threads when the application is run natively.
When using probes, Pin must be started with the PIN_StartProgramProbed() API.

Framework	***Supported architectures***	***Modes***	***Process attaching***
Pin	x86, x86-64, Itanium, ARM	JIT, Probe	Yes
DynamoRIO	x86, x86-64	JIT	No
Dyninst	x86, x86-64, PowerPC, SPARC	Probe	Yes

Table 1: DBI frameworks

4 Dyninst

Dyninst is a binary instrumentation framework that provides a machine independent interface to permit the creation of tools and applications that use runtime code patching.
Using its API, a program can attach to a running program, create a new bit of code and insert it into the program. The program being modified is able to continue execution and doesn't need to be re-compiled, re-linked, or even re-started. The next time the modified program executes the block of code that has been modified, the new code is executed in addition to the original code. The API also permits changing subroutine calls or removing them from the application program. [5]
Dyninst does this by providing a set of abstractions for programs and a simple way to specify the code to insert into the application. To generate more complex code, extra (initially un-called) subroutines can be linked into the application program, and calls to these subroutines can be inserted at runtime via this interface. These routines can be either statically linked, or loaded at runtime as part of a dynamic library.
Although this API can be used directly by programmers, it is primarily aimed at tool builders. As a result, the interface to code generation, based on ASTs[1], is convenient for tool builders, yet somewhat clumsy for hand construction.

4.1 How it works

1 Abstract syntax tree

The basic operations on the application process by the mutator process employ the same operating system services used by debuggers (e.g., ptrace, /proc filesystem, etc.). These services provide a way to control process execution, and to read and write the address space of the application program. In addition, a dynamic linked library that contains utility functions and two large arrays is loaded into the application to be instrumented. Both arrays are used for dynamically allocating small regions of memory. One of the arrays is used for instrumentation variables, and the other to hold instrumentation code. Both of these arrays are managed as heaps by the mutator process to provide dynamically allocated storage for runtime code generation.
In order to generate code, Dyninst translates the snippet into machine language code in the memory of the mutator process, and then copy it into the array in the application address space. When inserting instrumentation, Dyninst carefully modifies the original code to branch into the newly generated code using short sections of code called trampolines. Figure 3 shows the structure of a trampoline and its relationship to the instrumentation point.

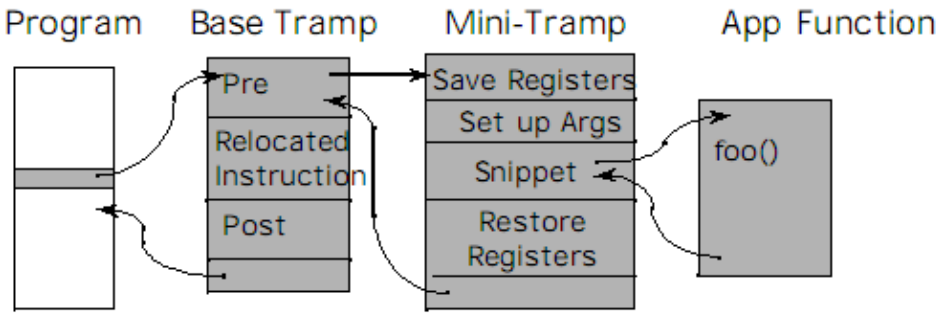


Figure 3: Inserting code using trampolines in Dyninst

Trampolines provide a way to get from the point where we wish to insert the instrumentation code to our newly generated code. To do this, we replace one or more instructions at the instrumentation point with a branch to the start of a base trampoline.
The base trampoline code then branches to a mini-trampoline. The mini-trampoline saves the appropriate machine state (such as the registers and condition codes), and contains the code for a single snippet. At the end of the snippet, we place code to restore the machine state and to branch back to the base trampoline. The base trampoline then executes the instruction(s) that were displaced from the original code. If the snippet is to be inserted after the point executes (i.e., after a function call return), we can also insert a mini-trampoline here. Multiple snippets can be inserted at a single point, and they are chained together such that the end of one snippet branches to the start of the next one, and the final snippet branches back to the trampoline.

4.2 Features

- Insert instrumentation into a binary on disk and write a new copy of that binary back to disk. (This includes rewriting .so/.dll files)
- Process attaching / detaching.
- The API includes a simple type system based on structural equivalence. If mutatee programs have been compiled with debugging symbols and the symbols are in a format that Dyninst understands, type checking is performed on code to be inserted into the mutatee. [5]
- Batch insertion of instrumentation code into the mutatee (significant performance increase)

4.3 API

The API is based on abstractions of a program and its state while in execution. The two primary abstractions are points and snippets. A point is a location in a program where instrumentation can be inserted. A snippet is a representation of a bit of executable code to be inserted into a program at a point. Snippets can include conditionals, function calls, and loops.
The API is designed so that a single instrumentation process can insert snippets into multiple

processes executing on a single machine. To support multiple processes, two additional abstractions, threads and images, are included in the API. A thread refers to a thread of execution. Depending on the programming model, a thread can correspond to either a normal process or a lightweight thread. Images refer to the static representation of a program on disk. Images contain points where their code can be modified. Each thread is associated with exactly one image.
The API includes a simple type system to support integers, strings, and floating point values. Additionally, support for aggregate types including arrays and structures is provided. The interface allows manipulation of user defined types that exist in the target application to be modified. There is no way to create new types using the interface, but a specific tool built using the API can create new types as part of its runtime library that is loaded into the application.
An inherent part of an API to manipulate other processes is the need to react to events of interest that take place in the application process. There are two types of events that occur in an application process, events caused by the inserted code and events that occur as a result of the normal execution of the application such as process termination. To provide a uniform way to handle these events, Dyninst has defined a variety of callbacks that inform the mutator of events of interest in the application. In addition, there are functions to query if an event has happened. [4]

There are three main components to the interface.

- First, there are the classes used to manipulate code in execution. This group includes the **BPatch** class, and the **BPatch_thread** class.
  BPatch represents the entire Dyninst API library. There can only be one instance of this class. This class is used to perform functions and obtain information not specific to a particular thread or image. It is also used to define the callback functions to be invoked for specific application events.
  BPatch_thread operates on (and creates) code in execution. This class can be used to manipulate the thread (or the process, in case of a non-threaded application).

- Second, there are classes for accessing the original program and its data structures. These include the **Bpatch_image**, **BPatch_module**, and **BPatch_function** classes.
  BPatch_image is the abstraction that represents the program executable.
  BPatch_module represents a program module, which is part of a program's executable image. Generally, a module refers to a single source file in the original program. However, for many libraries (especially dynamically loaded libraries), the module abstraction is used to represent the entire library.
  BPatch_function represents a function in the application. There are methods to get the entry and exit of a function and use them as instrumentation points.

- Third, there are classes to construct new code snippets and insert them. These include the **BPatch_snippet** class, and the **BPatch_point** classes.
  BPatch_snippet is an abstract representation of code to insert into a program.
  BPatch_point's are locations in an application's code at which the library can insert instrumentation. Points can either be described symbolically (i.e. the entry point to a function), or by providing a virtual address in the program (i.e. instrumenting a specific statement or instruction).

A collection of instances of the class BPatch_snippet, and specific sub-classes that represent different types of code to be inserted represent the statements to be added to the application by the mutator. The following types of code snippets are available in Dyninst:

- **BPatch_variableExpr** represents a variable or area of memory in a thread's address space.

- **BPatch_arithExpr** is used for most two operand statements in code definitions. Arithmetic expressions cover a large class of operations including variable assignment, basic mathematical operations, and array references. Arithmetic operations are only supported for predefined types.

- **BPatch_boolExpr** defines a set of comparison operations between two variables. The operations are only defined on the base types

- **BPatch_gotoExpr** provides a simple form of branching within snippets using a goto expression. The goto expression permits a snippet to branch back to an earlier part of that snippet. By combining it with conditional statements it can be used to construct loops.

To generate more complex code, extra (initially uncalled) subroutines can be linked into the application program, and calls to these subroutines can be inserted at runtime using **Bpatch_funcCallExpr** snippets. [5]

5 DynamoRIO

DynamoRIO is a runtime code manipulation system that supports code transformations on any part of a program, while it executes. DynamoRIO exports an interface for building dynamic tools for a wide variety of uses: program analysis and understanding, profiling, instrumentation, optimization, translation, etc. Unlike many dynamic tool systems, DynamoRIO is not limited to insertion of callouts/trampolines and allows arbitrary modifications to application instructions via a powerful IA-32/AMD64 instruction manipulation library. DynamoRIO provides efficient, transparent, and comprehensive manipulation of unmodified applications running on stock operating systems (Windows or Linux) and commodity IA-32 and AMD64 hardware. [6]

5.1 How it works

DynamoRIO operates in user mode on a target process. It acts as a process virtual machine, interposing between the application and the operating system. It has a complete view of the application code stream and acts as a runtime control point, allowing custom tools to be embedded inside it.

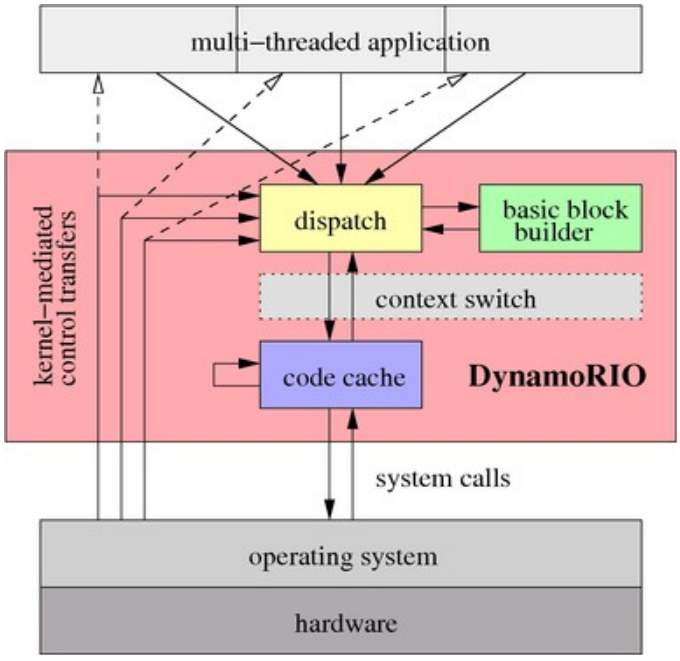


Figure 4: DynamoRIO structure

Much like Pin, DynamoRIO operates by shifting an application's execution from its original instructions to a code cache, where the instructions can be freely modified. DynamoRIO occupies the address space with the application and has full control over execution, taking over whenever control leaves the code cache or when the operating system directly transfers control to the application (kernel-mediated control transfers in figure 4).
DynamoRIO copies the application code one dynamic basic block at a time into its basic block code cache. A block that directly targets another block already resident in the cache is linked to that block to avoid the cost of returning to the DynamoRIO dispatcher. This is a feature like Pin's trace linking which leads to better performance.
Frequently executed sequences of basic blocks are combined into traces, which are placed in a separate code cache. DynamoRIO makes these traces available via its interface for convenient access to hot application code streams.
Configuration and execution of applications under DynamoRIO is handled by the drconfig, drrun, and drinject scripts on Linux.

5.2 Features

- **IA-32/AMD64 Disassembly Library:** DynamoRIO can be used as a standalone library for IA-32/AMD64 disassembly, decoding, encoding, and general instruction manipulation, independently of controlling a target application. For example, DynamoRIO's encoding routines take an instruction or list of instructions and encode them into the corresponding IA-32 bit pattern.

- **Instrumenting Multi-threaded Applications:** DynamoRIO provides some utilities through its API to facilitate development of multi-threaded clients and instrumentation of multi-threaded applications. DynamoRIO's API provides:

    - Memory allocation: both thread-private (faster as it incurs no synchronization costs) and thread-shared
    - Thread-local storage
    - Thread-local stack separate from the application stack
    - Simple mutexes
    - File creation, reading, and writing
    - Address space querying
    - Application module iterator
    - Processor feature identification
    - Extra thread creation
    - Symbol lookup (currently Windows-only)
    - Auxiliary library loading

- **Extensions:** DynamoRIO supports extending the API presented to clients through separate libraries called DynamoRIO Extensions. Extensions are meant to include features that may be too costly to make available by default or features contributed by third parties whose licensing requires using a separate library. Examples of extensions are symbol access and container data structures.

- **Persisting Code:** Decoding, instrumenting, and emitting code into the code cache takes time. Short-running applications, or applications that execute large amounts of code with little code re-use, can incur noticeable overhead when run under DynamoRIO. One solution is to write the code cache to a file for fast re-use on subsequent runs by simply loading the file. DynamoRIO provides support for tools to persist their instrumented code.

5.3 API

DynamoRIO exports a rich Application Programming Interface (API) to the user for building a DynamoRIO client. A DynamoRIO client is a library that is coupled with DynamoRIO in order to jointly operate on an input program binary.

A client's primary interaction with the DynamoRIO system is via a set of event callbacks. Event interception functions, if supplied by a user client, are called by DynamoRIO at appropriate times. These events include the following:

- Basic block and trace creation or deletion
- Process initialization and exit
- Thread initialization and exit
- Fork child initialization (Linux-only); meant to be used for re-initialization of data structures and creation of new log files
- Application library load and unload
- Application fault or exception (signal on Linux)
- System call interception: pre-system call, post-system call, and system call filtering by number
- Signal interception (Linux-only)

Typically, a client will register for the desired events at initialization in its dr_init() routine. Each event has a specific registration routine (e.g., dr_register_thread_init_event()) and an associated unregistration routine.

DynamoRIO provides two events related to fragment creation: one for basic blocks and one for traces (**dr_register_bb_event()** and **dr_register_trace_event()**). Through these fragment-creation hooks, the client has the ability to inspect and modify any piece of code that DynamoRIO emits before it executes. Using the basic block hook, a client sees all application code. The trace-creation hook provides a mechanism for clients to instrument only frequently-executed code paths.

In addition, the API includes several higher-level routines to facilitate code instrumentation. These include the following:

- Routines to insert clean calls to client-defined functions.
- Routines to instrument control-flow instructions.
- Routines to spill registers to DynamoRIO's thread-private spill slots.
- Routines to quickly save and restore arithmetic flags, floating-point state, and MMX/SSE registers.

Probe mode is not supported in DynamoRIO anymore. Although the Probe API is still present in the source code it is not documented and not maintained anymore.
Generally, DynamoRIO requires the tool writer to work on a lower level of abstraction than Pin, since the instruction encoding library of DynamoRIO is used to make function calls at run time in the instrumentation code.

Earlier versions of DynamoRIO would not start instrumenting the target program from the beginning and would miss some of the initialization code of the target program. In recent versions an -early flag is provided which will make it possible for the DynamoRIO tool to inspect the initialization part of the target application as well.

6 Comparison

In this section, we will compare the performance of Pin and DynamoRIO. The comparison is done using NAS Parallel Benchmarks(NPB)[7] to test the performance overhead of Pin and DynamoRIO. The NAS Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers.

For this purpose we will use the class S of the following five specifications from the OpenMP-based implementations of NBP3.3:

- IS - Integer Sort, random memory access
- EP - Embarrassingly Parallel
- CG - Conjugate Gradient, irregular memory access and communication
- MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive
- FT - discrete 3D fast Fourier Transform, all-to-all communication

The instrumentation tool is a memory trace program written in Pin and DynamoRIO.

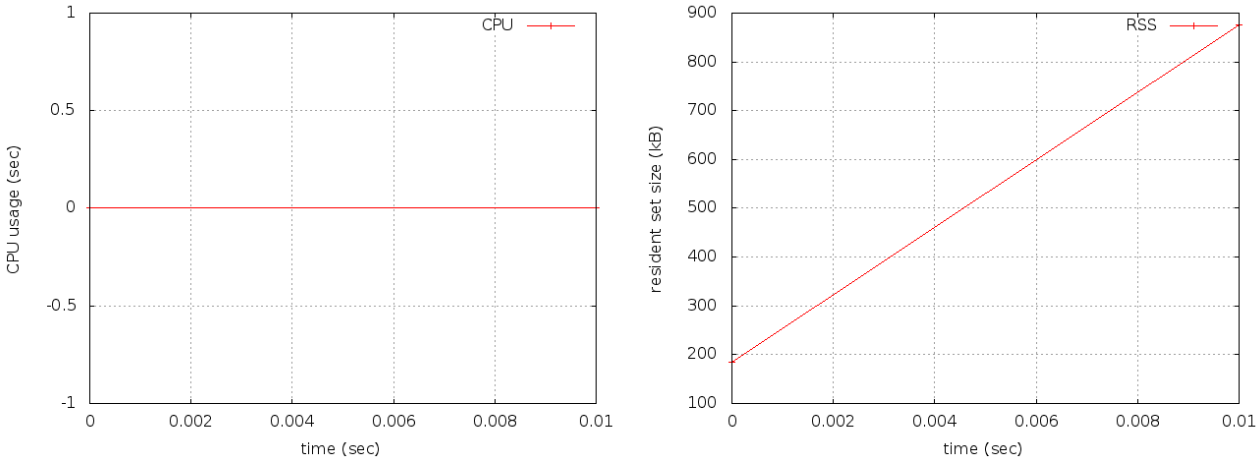
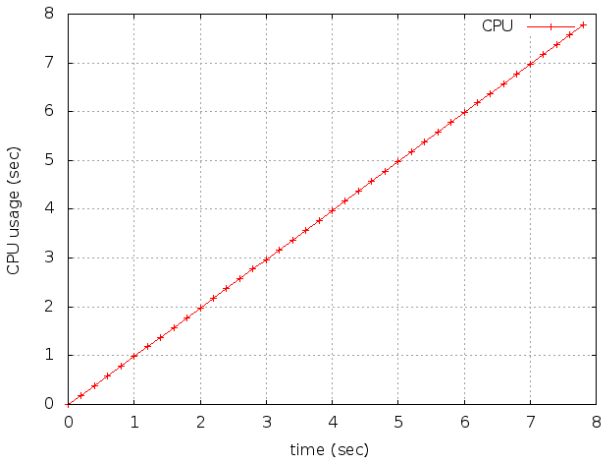


Table 2: IS – Uninstrumented



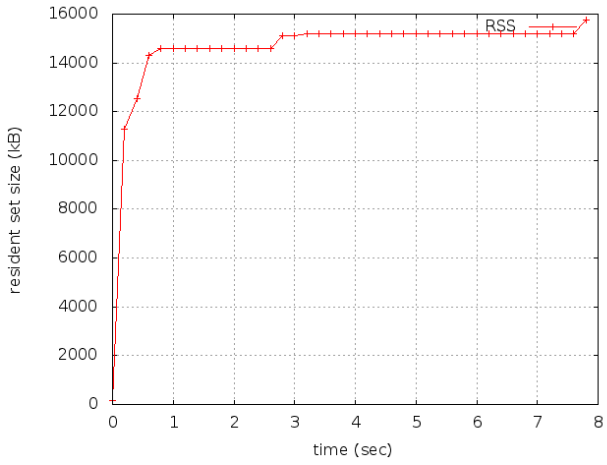
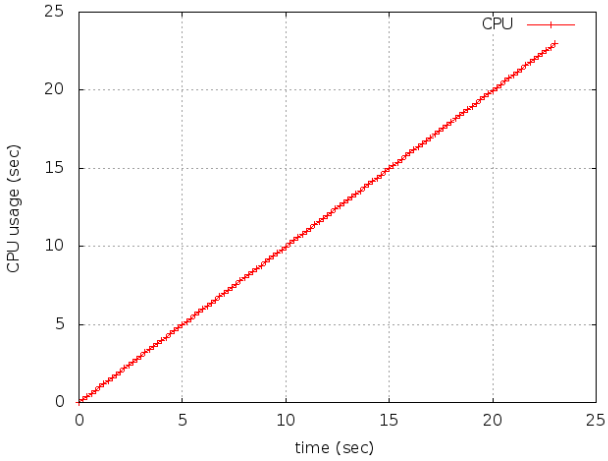


Table 3: IS – Pin



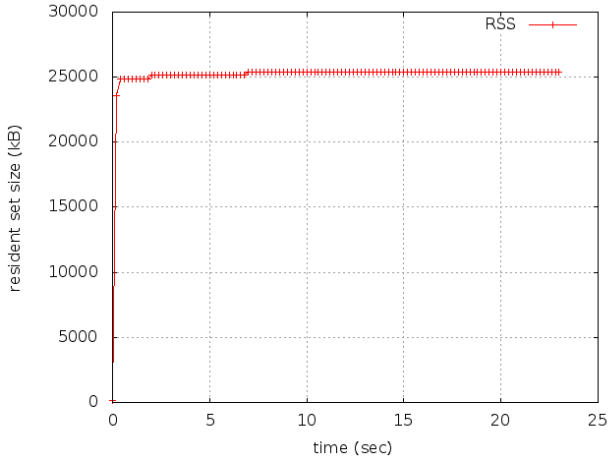
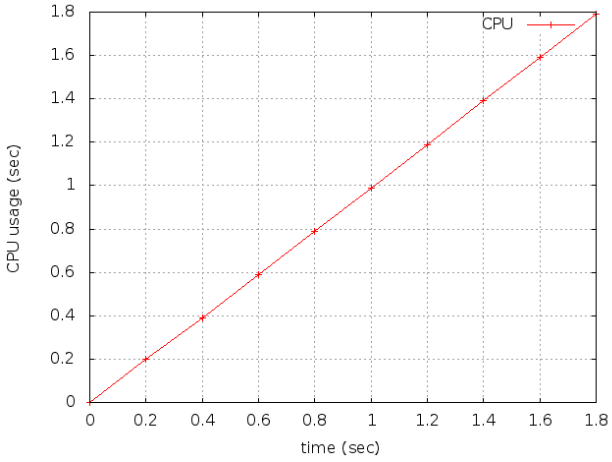


Table 4: IS – DynamoRIO



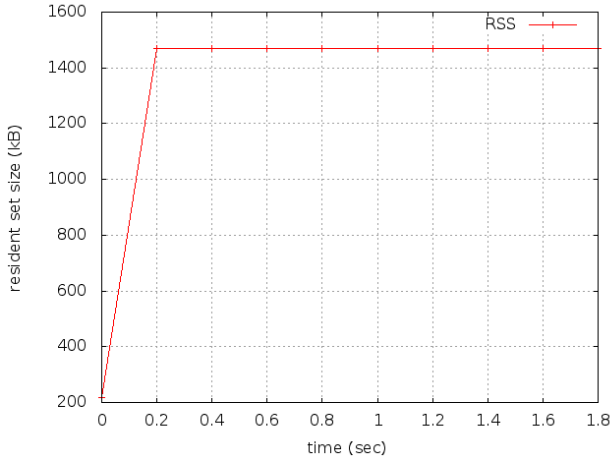


Table 5: EP – Uninstrumented

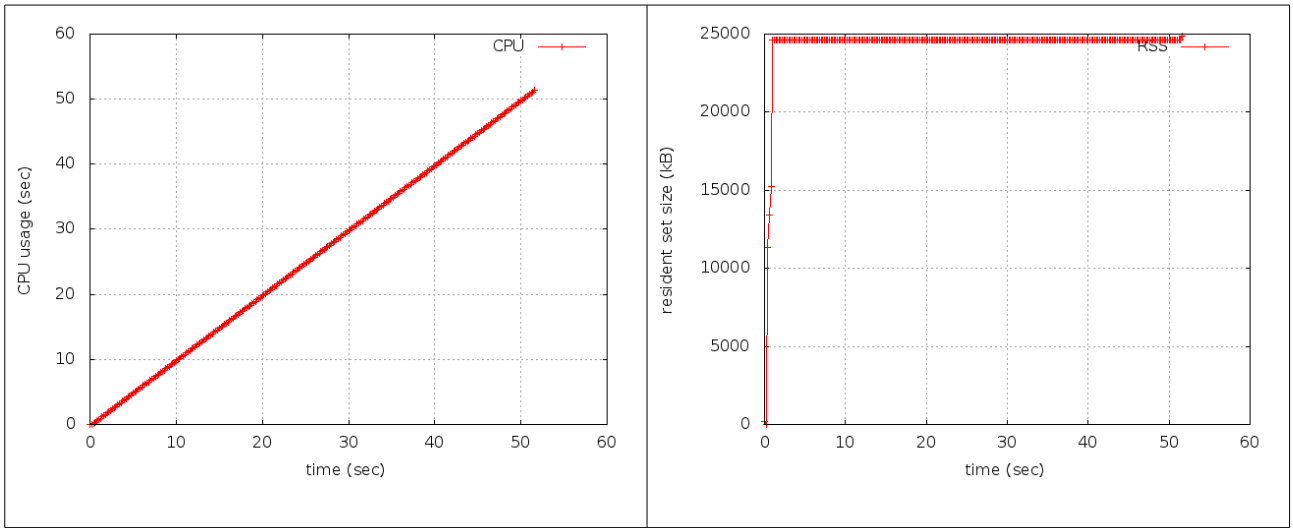


Table 6: EP – Pin

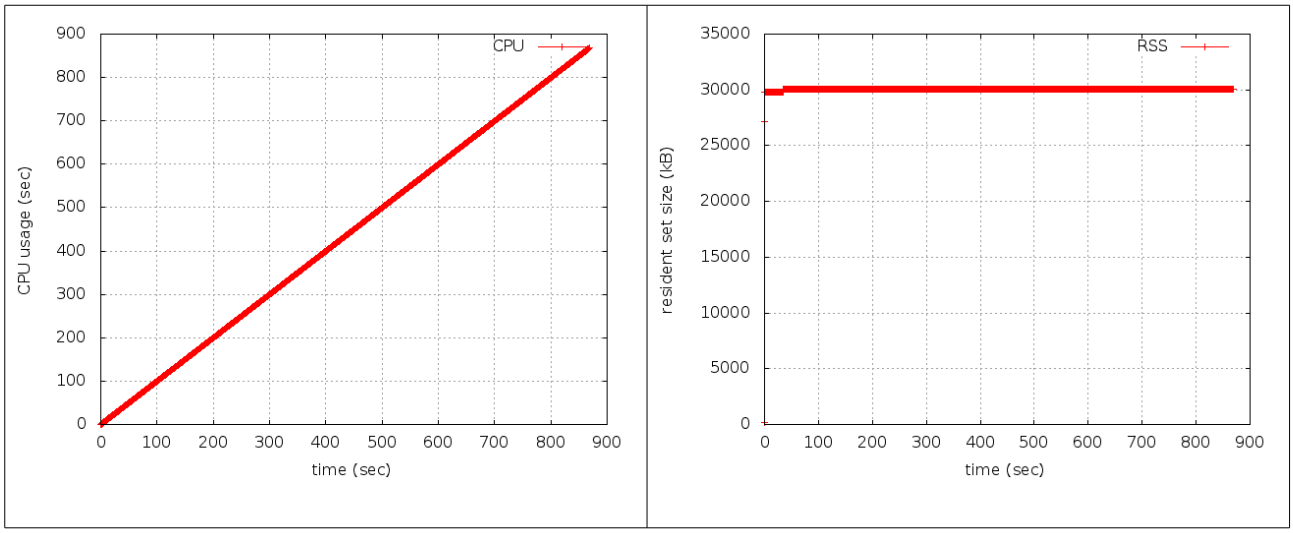
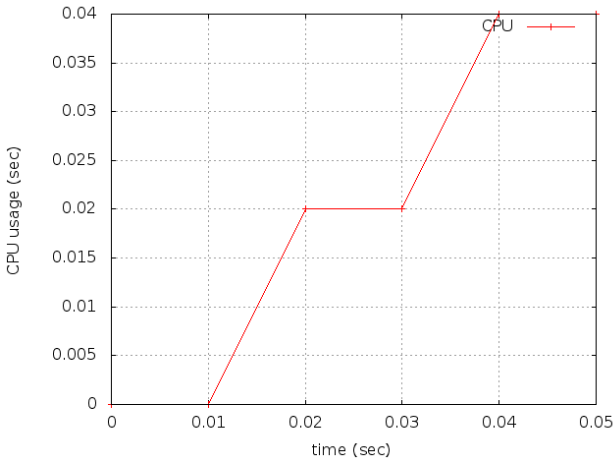


Table 7: EP - DynamoRIO



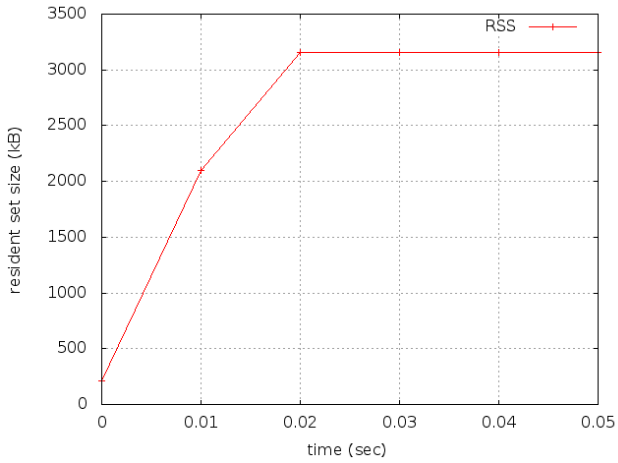
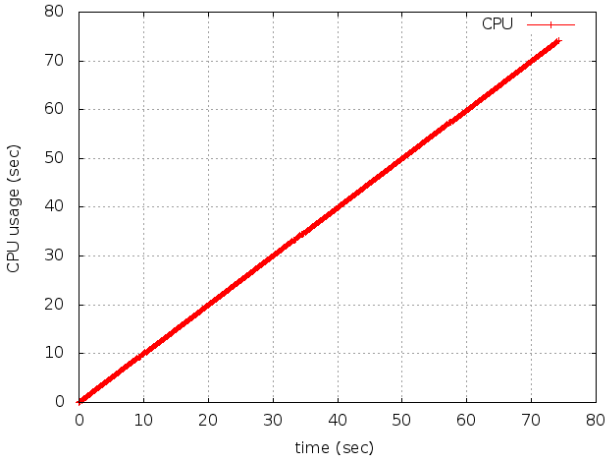


Table 8: CG - Uninstrumented



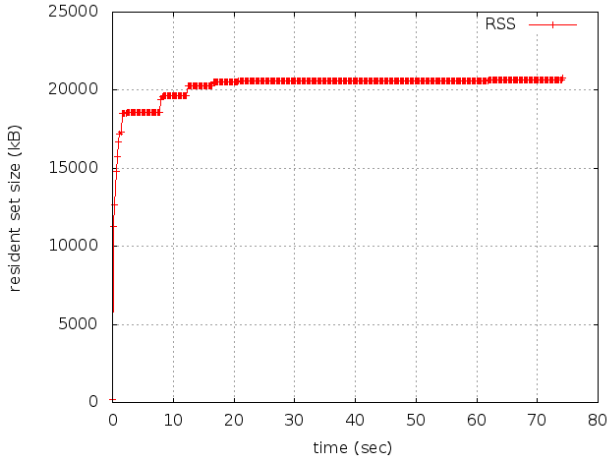
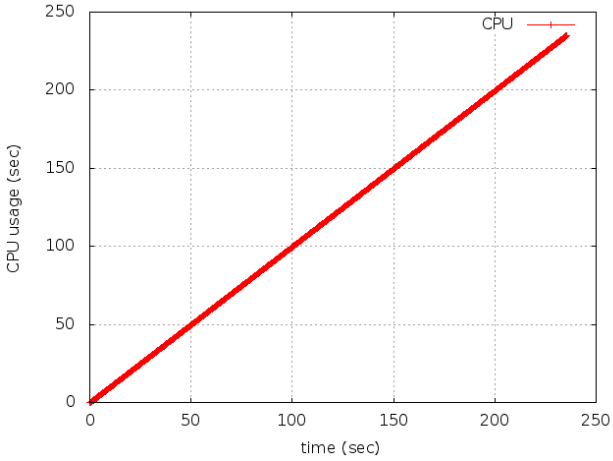


Table 9: CG – Pin



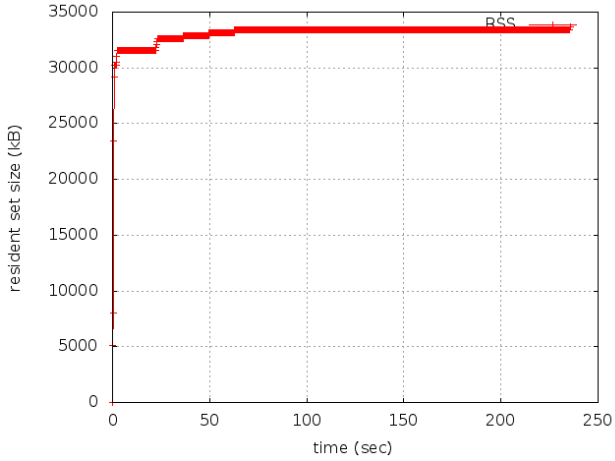


Table 10: CG – DynamoRIO

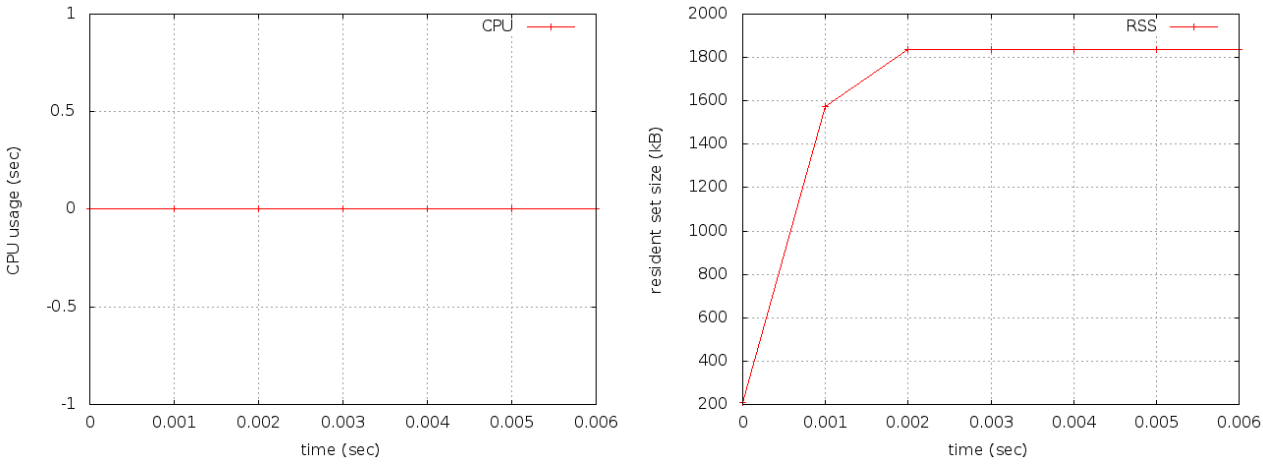


Table 11: MG – Uninstrumented

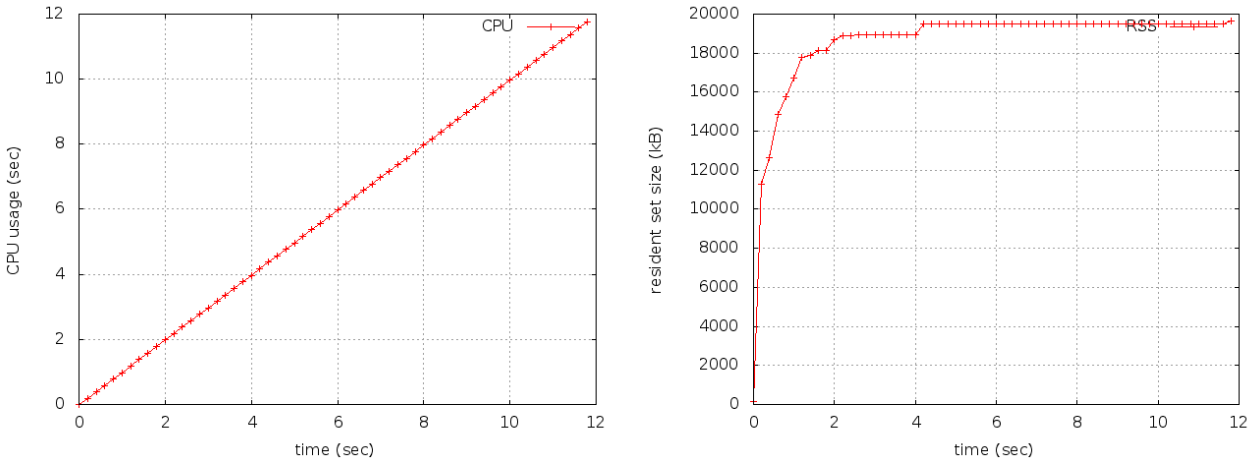


Table 12: MG – Pin

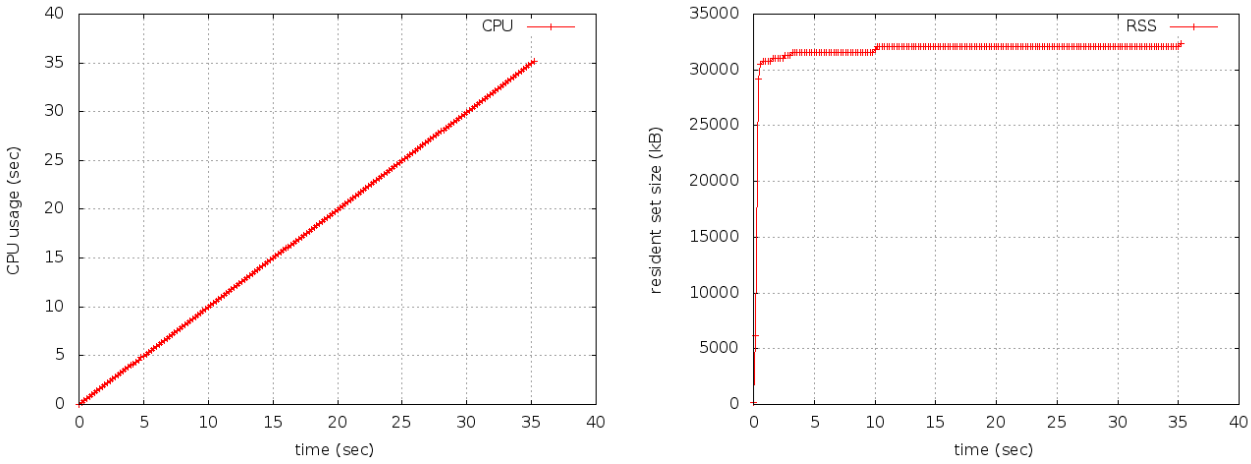
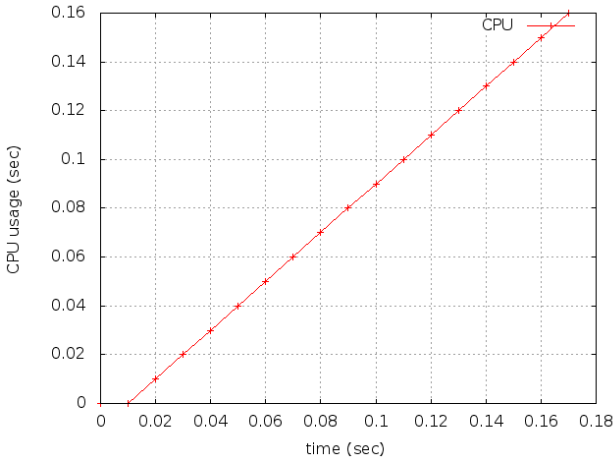


Table 13: MG – DynamoRIO



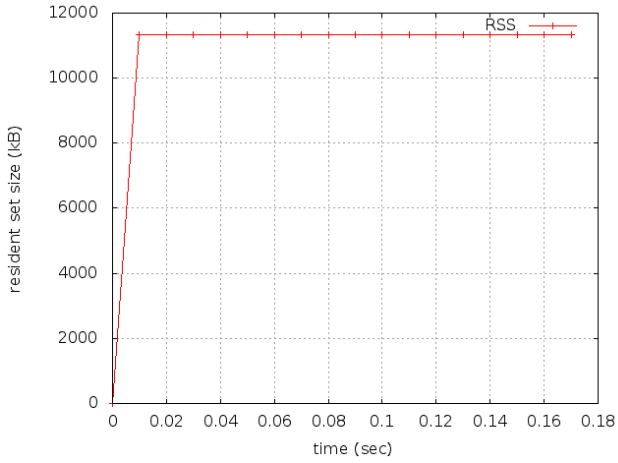
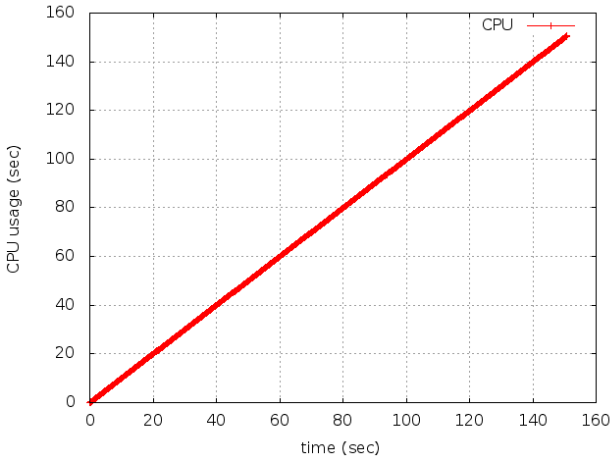


Table 14: FT – Uninstrumented



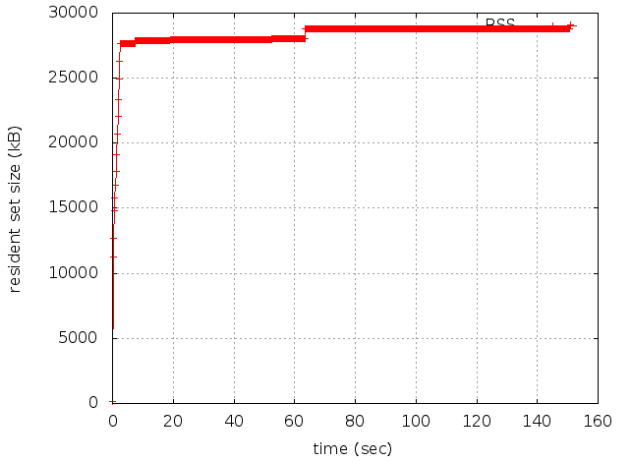


Table 15: FT – Pin

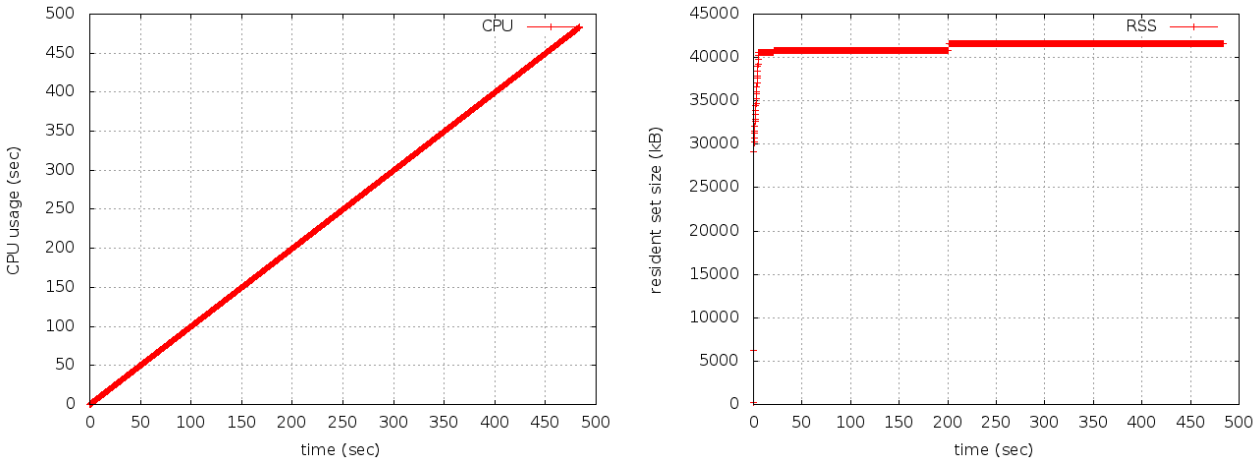


Table 16: FT – DynamoRIO

In all cases Pin has less overhead on memory and CPU usage of the instrumented program than DynamoRIO.

7 References

1 Pin 2.12 User Guide, http://software.intel.com/sites/landingpage/pintool/docs/53271/Pin/html/

2 Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation, www.cs.virginia.edu/kim/tortola/papers/pldi05.pdf

3 Pin tutorial CGO 2012/ISPASS 2012,http://software.intel.com/sites/default/files/m/d/4/1/d/8/pin_tutorial_cgo_ispass_2012.ppt

4 An API for runtime code patching, http://www.dyninst.org/sites/default/files/apiPreprint.pdf

5 Dyninst Programmer's Guide, http://www.dyninst.org/sites/default/files/manuals/dyninst/dyninstProgGuide.pdf

6 DynamoRIO Documentation, www.dynamorio.org/docs/

7 NAS Parallel Benchmarks, http://www.nas.nasa.gov/publications/npb.html