IMPERIAL COLLEGE LONDON

A Virtual Time Performance Engineering Tool

Author:
Giovanni CHARLES

Supervisor:
Dr. Anthony J FIELD

Contents

Abstract

In this report we investigate the use of speculative profiling as a performance engineering tool for binaries. We introduce two frameworks for the performance analysis of binaries through instrumentation. We combine these frameworks with a virtual time execution engine to create a binary profiler with predictive capabilities.

I would like to thank my family and close friends for their tolerance of my obsession. Many more thanks to Tony Field for fueling that obsession.

1. Introduction

Many modern day systems are performance critical. The increasing demand for processing power has pushed web servers, mobile devices and desktops to their limits. Performance engineers spend a great deal of effort in making sure our systems are up to the task.

The typical cycle of a performance engineer is to understand a system, hypothesise and then optimise. For example, a performance engineer, working in a team of developers in a high end 3D Game studio, may be responsible for optimising the rendering of a landscape. Suppose that during her investigations she finds a section of the code that inefficiently loops on an expensive I/O call.

At this point she hypothesises about the possible solutions. She could design a cache, parallelise the loop or even change the code algorithmically. Any of these avenues could have far reaching knock on effects on the complex, interconnected game. Even an engineer who is familiar with the code base could not predict the final outcome of any change. Furthermore, all the tools in the industry do not add to her understanding of the codebase, they simply identify the bottleneck she has already found.

A Virtual Time Framework (VTF) is a tool that would solve this problem. Our performance engineer would benefit from a tool that could simulate the effects of her changes so that she can make an informed decision. A VTF is able to do this by virtualising the passing of time in execution. It then produces a speculative profile that gives a better insight into the system.

1.1 Contributions

In this report we introduce:

- An instrumenting framework for profiling C/C++ binary function calls and controlling threads of execution
- A dynamic instrumenting framework for adaptive profiling of C/C++ binaries.
- A complete integrated system for virtual time profiling of C/C++ binaries.
- An insight into tuning adaptive profiling to efficiently produce virtual profiles.

These contributions will form a profiler that is able to speculatively profile a multi-threaded C/C++ binary.

2. Background

2.1 Concurrency in C/C++

There are several ways to achieve concurrency in C/C++. Systems may opt for standard or third party implementations of concurrent platforms. Heavyweight systems may choose to distribute execution across several machines using network communications like MPI [23]. More conservative systems may parallelise on a process level.

The most lightweight of solutions is to use threads. A thread of execution has minimal overheads, shares the same address space as its creating thread and can run concurrently on the same CPU. There are several thread platforms for C/C++ including *Boost* [7], *OpenMP* [26], *Concurrency* [9], C11/C++11 threads. These threads can be synchronised with many different systems systems, like *SysV*, *Linux futexs*.

This paper will focus on systems conforming to the *Pthread* standard and running on Linux. In the vast majority of cases a multi-threaded program will use the POSIX thread standard (Pthreads), or one of the many frameworks based on this standard. This assumption will allow us to reason about many of the systems in production today.

2.1.1 Pthreads

Pthreads in Linux are implemented in the Linux's Native POSIX Thread Library (NPTL). It uses a linux process to implement each thread. It is designed to efficiently share memory and synchronise through system calls, as described in [14]. To use these threads, a program must load the pthread shared library, libpthread.so, and create threads by calling it's API:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine)(void*), void *arg);
```

This call will spawn a thread. The outcome is that the new thread handle is stored in *thread*, the attributes specified in *attr* are applied to the new thread and its execution begins at *start_routine* with the sole argument *arg*. Thread attributes can be used to configure threads to detach from the parent or change the guard buffer or stack size. Once *start_routine* returns or *pthread_exit* is called the thread will be destroyed.

2.1.2 Synchronisation

Pthreads offers the following primitives to manage synchronisation. We can explain these primitives with references to the example code at appendix A.

Locks

These allows programmers to enforce mutual exclusion on sensitive sections of code.

Once a thread acquires a mutex (line 24) any thread that attempts to acquire the same mutex (line 53) will block until the owner unlocks said mutex (line 34).

Monitors

When a thread calls *pthread_cond_wait* (line 56) with a lock and a condition, the thread will release the lock and block until the condition is known to be true. Live threads can signal a single thread to wake up (line 32) or broadcast to all waiting threads to wake up. A timed variant exists, *pthread_cond_timedwait*, in which a thread will be woken after a certain time regardless of the condition.

Once the thread is awake it will attempt to reattain the lock it released on entry before continuing execution.

Join

A thread that spawns another threads may want to wait for its execution to end.

A join (line 84) will block a parent thread until a given thread is destroyed.

Yield

A yield signals the scheduler remove the calling thread from the cpu. NPTL will place the calling thread at the back of the run queue so that it will execute after all other threads have had the chance to execute.

Sleep

(line 37) blocks the calling thread for a given number of seconds.

2.2 Analysis tools for C/C++

Multi-threaded programs can become complicated very quickly. A variety of tools have have been created to give a deeper understanding of the run-time execution of a system. For the remainder of the report we will call this action profiling. The type of multi-threaded program we wish to analyse will be referred to as the input. We will refer to the program that wishes to analyse the input as the profiler.

2.2.1 Control flow graphs

An automated analysis of these programs requires a meaningful representation of a program to work with. Control flow graphs show the possible ways in which a thread could progress through a program. These graphs can be used to automatically identify sensitive code and other points of interest.

Many code parsers exist that take inputs ranging from source code to binary executables and produce control flow graphs:

Compilers

Extensible compilers like the GNU C Compiler [15] and Clang/LLVM [8] provide interfaces to their intermediate representations, some of which serve as control flow graphs.

Source level analysis like this is very attractive since there is a lot of useful meta information available at this stage in the compilation process.

Binary analysers

A less popular approach to analysis of the input is to parse the binary executable.

With binary analysis the input size is smaller so there are fewer overheads with the analysis. Also, at the machine code level you get a more precise picture of what will happen on your hardware.

However, this has removed the profiler from the high level constructs that programmers know and understand. This means the profiler has to go through extra effort to reverse engineer these constructs before reporting back to the programmer. For example, the function entry and exit points that may exist at the source level may change during compilation at the compiler's discretion.

GNU provides several tools to analyse Linux binaries [6].

Dyninst [2] have built a framework on top of these tools called the *Parse API* to provide a comprehensive system for analysing the control flow of a binary.

2.2.2 Ptrace

In order to extract run-time information from an input, the most widely used method is *ptrace*. *Ptrace* is a Linux system call that allows a profiler to examine or control the input. The call has the following signature:

```
long ptrace(enum __ptrace_request request, pid_t pid,
  void*addr, void *data);
```

The type of requests that can be made allow control over the input's registers, memory and flow of execution at any point. Each request is explained in detail in the extract of ptrace.h in appendix B.

The full power of *Ptrace* can be seen in *gdb* [16], a popular Linux debugger. We can use some of the features of *gdb* to convey the potential of *ptrace*. In depth details of *gdb* can be found at [11].

Signal handling
gdb will first use the ”set siginfo” request to register itself as the handler for any special OS signals the input my receive.

Breakpoints
To set a breakpoint, *gdb* would use the ”read data” and ”poke data” requests to replace an instruction in the input with one that breaks execution, for example a trap or a zero division. When this instruction is executed, the OS will give control to *gdb* since it is registered as the handler.

Stepping
For finer control over the execution of the input, *gdb* can progress through a paused input one instruction at a time. This is done with the blocking requests like ”single step”. When single step is requested, *gdb* will block until the operating system has executed one instruction of the input and paused its execution again.

Expression evaluation
A very useful feature of *gdb* is its ability to evaluate expressions in the input process' address space. The user can use commands like `print foo->bar[5*sizeof(elem)]` to debug the state of the input. This feature is implemented with an interpreter that uses the ”read data” request to retrieve data from the input, word by word, so that it can evaluate the result of the expression and print it out to the user.

Ptrace's appeal is that it can be used to analyse any binary without recompilation or manipulation of its code. Unfortunately, analysis of the input from a remote address space would incur significant speed overheads from signal handling delays or word-by-word data manipulation with ”poke data”.

2.2.3 Instrumentation

An alternative approach is to augment the code of the input so that it reveals information about its own execution.

Several methods of instrumentation exist:

2.2.4 JIT instrumentation

With JIT instrumentation has the following process:

1. Read a block of the input's binary into it's address space
2. Interpret this block into an intermediate language

3. Perform runtime analysis adds instruments to the code
4. Recompile the intermediate code to native code
5. Execute the native code
6. Repeat

A profiler could intercept this process at step 3 and perform its own anaylsis of the input. Three of the most popular JIT instrumenters are PIN [21], DynamoRIO [12] and Valgrind [25].

The upside to this method is the unlimited control given to the profiler. Since it is responsible for executing the input, it can observe the full state of the system, arbitrarily control its execution. The costs of interpreting the binary have been highly optimised by PIN and DynamoRIO for speed and by Valgrind for intermediate language quality. PIN and DynamoRIO have elaborate caches and instruction schedulers to skip out any of steps 2-4 when they are not needed so the execution overhead is minimised, the specifics of each system are described in more detail in [21, 12]. Valgrind on the otherhand spends more time on step 3 so that the high level programming constructs can be reverse engineered from the binary. An example of this is shadow memory, which keeps track of every high level variable in memory so that it may be analysed [24]. Systems like Valgrind cause too much of a performance penalty to be useful for highly time sensitive profilers. Profilers that are built on these systems, like Callgrind, have value in the more detailed profiles they can create with cache and branch prediction simulation.

Even for the faster instrumenters this control comes at a high cost. The profiler would share the same address space as the input code. This means analysis code will have to take great care not to interfere with the input code's memory. For example, dynamoRIO does not allow analysis code to use the Pthreads library and all heap allocation must happen through special methods that separate the profiler's address space from the input's address space. This makes integration of JIT instrumenters with more complicated profilers a difficult process.

Less intrusive methods to edit the host environment may produce more representative results. Many developers are of the opinion that to best understand the input, profilers should inspect it while minimising the effect they have on it and the environment. Also for native binaries, allowing the input to run directly on the host OS will be the environment closest to the release environment.

2.2.5 Binary rewriting

At the other extreme is binary rewriting. In this solution the profiler works with "trampoline code" to analyse run-time execution. Trampoline is a term we will use to describe code that is inserted into a binary to change the flow of execution without breaking the expected behaviour of the input. The profiler prepares the input for execution by:

1. Modifying the input's initialisation to load a shared library containing the trampoline code.
2. Parsing the binary in to a control flow graph

3. Identifying the parts of the code it wants to change
4. Replacing instructions in the input code with jumps to the trampoline code.

The input code will then be run. A trampoline is illustrated in figure 2.1.

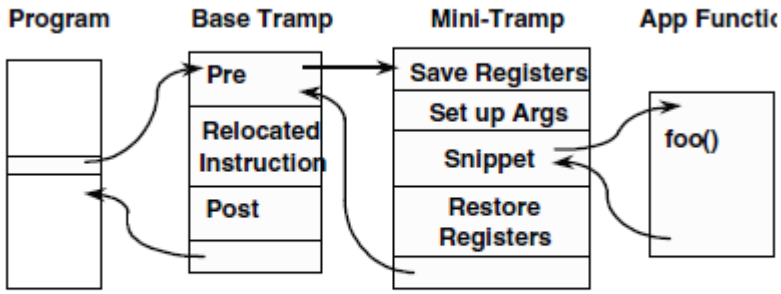


Figure 2.1: To insert code into a program, the state must be saved before the new code is run. The program state must then be restored before the program continues execution. Image from [29]

This sacrifices the flexibilities that JIT instrumentation offers 2.2.4 to drastically reduce the impact on execution speed [1]. You can attribute are reduced by not interpreting the code and the observer effect is reduced by allowing the input to run natively. Implementations of binary writing are described here [19, 2]

Dynamic rewriting

Also called run-time injection is an extension to binary rewriting in which the instruments can be added and removed during execution of the input code. To achieve this the profiler continues to run alongside the input code and uses ptrace, 2.2.2, to make run-time changes to the input's instruction code.

2.3 Dyninst

Dyninst is a collection of tools that can be used for binary and dynamic rewriting.

ParseAPI

The parse API is used to build instruction set agnostic control flow graphs from binaries. It provides abstractions like:

- modules (shared libraries, system libraries, object code ...)
- functions
- function calls

- branches
- loops
- ...

This library will be used by the analyser to parse the input code.

PatchAPI

Patch is an interface built on top of the previous framework to allow for manipulation of the graph. This is done with the *snippet* abstraction which represents a piece of code that can be inserted into the code or changed. Examples of snippets are:

- function calls
- literal strings and numbers
- arithmetic expressions
- sequence of snippets
- ...

The instrumenter will manage the creation of these snippets so that they can interact with CINE's collector.

ProcControlAPI

ProcControl acts as a wrapper to *ptrace*.

It would allow users to set callbacks for events like OS level events like:

- thread creation and destruction
- POSIX signals
- process forks

However, its main use will be to execute code in the input program's address space and to remove instruments at run-time for adaptive profiling.

BPatch

This is a top-level API unifies all other Dyninst APIs. It is useful as a start point however its functionality and performance is limited. For more complicated operations the former APIs should be used.

2.4 Profilers

The profilers in industry today use some of these methods to output various types of profiles. The profiles produced generally come in two forms:

flat

A table showing the number of calls and the total time spent inside each procedure, often as a percentage of the full execution time

call graph

A graphical representation of the flat profile showing the control flow of execution

Approaches for attaining these profiles are split into two main catagories, Sampling and Instrumentation. Comparisons between the two classes have been mainly speculative and takes place almost solely on discussion boards and blogs. Some of the differences between the types of profilers and the profiling options available are discussed in more detail here [3].

2.4.1 Sampling

Most of the profilers you see will use sampling methods. The idea is that the profiler regularly samples the input's progress through its execution. When the profiler terminates it is then able to statistically determine where the input spent most of its time and even the most frequent control flows in the system. A popular implementations of a sampling profiler is OProfile [20]

In this method the profiler and the input execute concurrently on separate processes. The profiler makes regular calls to *ptrace* to sample the input code's progress through execution. The calls to *ptrace* reveal the current instruction and a stack trace at that point of the input's execution. This information is recorded and compiled into a fine grained report sometimes giving line by line sample rates, a call graph profile or the common flat profile.

This technique is favoured by C performance engineers. Profiles made with this method can produce fine grained, line by line, information. The input code requires no preparation and can even be profiled after it has started running, which is useful for continuous processes like web servers.

2.4.2 Instrumenting

Conversely, instrumenting profilers output a complete profile of execution. They time how long each procedure in the input code takes to execute and piece together a complete profile on termination.

This is generally done by adding instruments to the entry and exit points of functions to measure the time for execution of the function body. Various methods exist for doing this: gprof extend gcc to allow users to compile in the instruments, automatic source level instruments can be added and some experimental profilers have explored binary rewriting [3, 27]. The latter projects are undocumented, unevaluated and the industry adoption is generally small.

The advantage over sampling is that the profiler can select the important points of interaction with the input program so that it can reduce its observer effect. However, there are often cases where instrumentation code is called too often altering the input's execution too much and producing unrepresentative profiles. For example, an input may make a call to an accessor method that takes very little time to run like so:

```
int Foo::getBar(){
  return this._bar;
}

...

for(...){
  foo.getBar();
}
```

The profiler would insert instruments at the start of getBar and before the return of the *_bar* field. The time the input spends timing the accessor method will cause a significant overhead which is compounded by the fact that it is called in a loop.

2.5 Virtual time

Virtual time execution is not a new idea. It was proposed in [18] as a method for controlling a distributed system's concurrency or synchronisation by organising each process's time sensitive events onto a single "virtual timeline".

The value of this virtual timeline is that it provides a representation of the system that can be used to efficiently analyse the state of the system or predict the effect of changes in execution time.

2.6 Virtual time system

A virtual time system allows processes to execute on independent virtual timelines and enforces the behaviour of a conventional system. It does this by applying a local control mechanism to each process and creating a global control entity.

2.6.1 Local control

Since each process executes according to its own virtual clock, it may be out of sync with any other process. During execution, when a process arrives at a time sensitive event it must record its virtual time and raise a timed event.

On recieving a timed event a process will alter its execution to reflect its expected behaviour. This is trivial for a future event since the process will handle the event when its own virtual clock matches that of the event. Past events come at the penalty of reverse execution and unraising of events, illustrated in figure 2.2

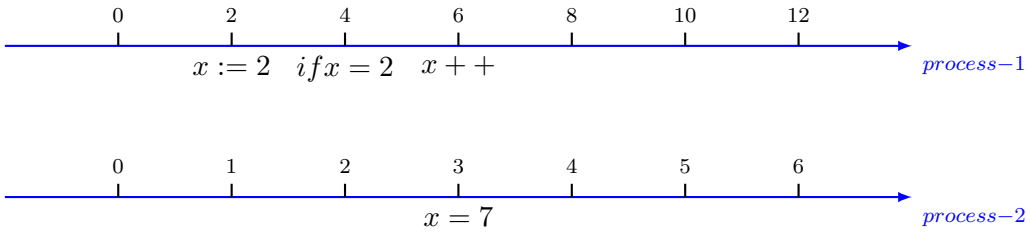


Figure 2.2: An example of when a process receives a past event. *Process-1* is executing twice as fast as *process-2*. When *process-1*'s local time reaches time 6, it will receive *process-2*'s event from time 3. *process-1* would have to roll back execution to before the sensitive instruction at time 4. This could also be avoided by preemptively pausing *process-1* at time 4.

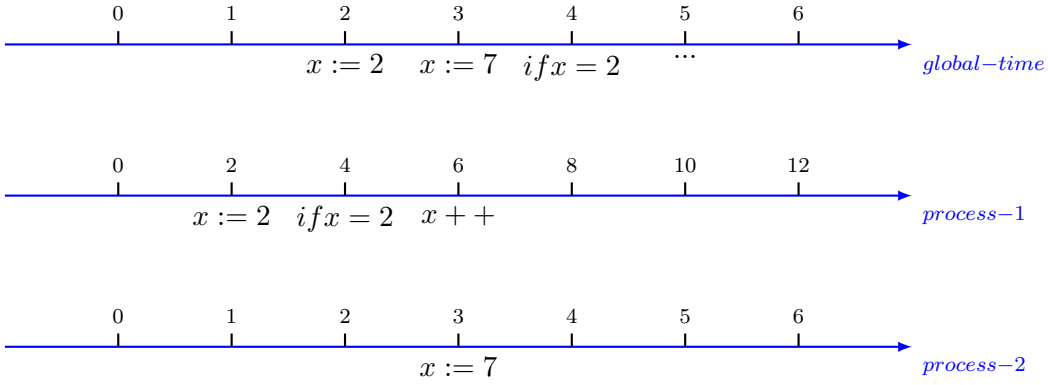


Figure 2.3: A global time mapping. Notice how the $x++$ event does not appear on the global timeline because it was unsent and never re-sent after *process-1* received the $x := 7$ event

2.6.2 Global control

Global control is needed to ensure progress is made and to detect termination of the overall system. To do this it maintains a global timeline, which is essentially a mapping of all the timed events. This has a useful side effect in that the global control can be used to give a real time report of execution, as shown in figure 2.3

2.7 Virtual time for Multithreaded systems

In the context of a parallel pthread program, a virtual time system could be used see the effects of changes in a thread execution time on the global timeline of a system.

This can be done by simply applying the model in 2.8 to a multithreaded system. Theads would be represented by the processes, synchronization primitives will take the form of timed events and

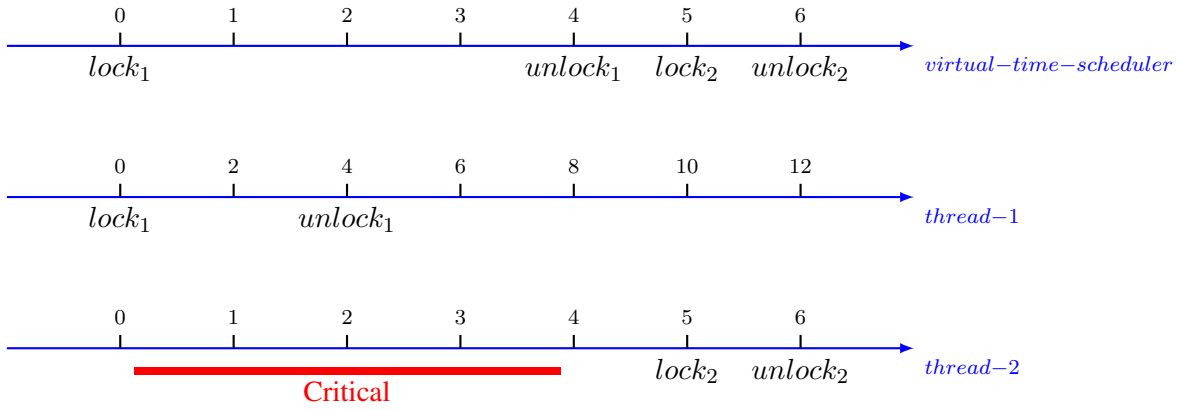


Figure 2.4: An application of virtual time to a synchronising system. A soon as *thread-2* receives the timed lock message from *thread-1* at time 0 it will know not acquire the shared lock. At time 2 when it receives the timed message that the lock will be released at time 4, *thread-1* will know that it has to sleep until this time if it wants to acquire the lock, even if in reality the lock may be available.

an advanced scheduler would be required to enforce the global control mechanism described in 2.6.2.

Thread local time would then be allowed to speed up and slow down independently from the rest of the system. The scheduler would be able to translate the events on each thread to a real time that could be used for a prediction of a real execution time. A simple example is shown in figure.

2.8 Virtual Time Framework

The Virtual Time Framework (VTF) [4] is an implementation of such a system for concurrent Java programs. This system lets a performance engineer configure the virtual timeline of arbitary threads and predict the effect on the overall execution of a program. This allows developers to see the gains of particular performance improvements on complicated systems before investing time and money into reengineering.

2.8.1 Features

Given a Java input program and arguments the VTF can execute the program in virtial time while performing some of the following timeline manipulations. It will then produce a flat or call graph profile similar to those produced by conventional profilers.

Method speedup

A user can specify a method of a class in the input program and a speedup value. During the method's execution, the VTF will change the rate of time to reflect the speed up. The knock

on effects of this speed up to other parts of the program will naturally be modelled by the global timeline.

This can be useful in predicting the effects of implementing a caching mechanism, changing to a different data structure or implementing a different algorithm.

Method modelling

For less predictable methods, a performance model can be provided. In this case, the VTF will use the model to compute the thread's virtual timeline during the execution of the method.

We could use this to predict see how different distributions of the chaotic response times from a remote system would effect our Java program.

I/O

The VTF has functionality to model the effects of different types of I/O on the program.

VTF has a mechanism to predict I/O response times so that it can continue scheduling global timeline while threads wait in I/O.

By default this is done empirically. VTF observes the I/O for a warm up period. It separates the various I/O calls by type on invocation by using the stack trace. Once the warm-up period is over, VTF will apply any of a variety of statistical models to predict the times for future I/O calls. When a thread returns from I/O, VTF must adapt the global timeline to take to simulate the thread returning at the observed time.

This system has been extended to allow for I/O scaling. This is when VTF simulates the effects of having longer than normal I/O times.

Multicore simulation

On top of this, it can simulate the execution of the input on a CPU with a different number of cores. For this, VTF maintains a virtual timeline for for each core on the system and schedules threads to free cores.

This would be useful on server farms where they would like to have an idea of the performance of programs on a CPU before buying a large number of machines.

Adaptive profiling

On computing the execution time of a method, if it is deemed to be insignificant it can be ignored for the remainder of the execution. [4] shows this to reduce the overheads mentioned in 2.4.2.

2.8.2 Summary of internals

VTF is separated into two sections, the Virtual EXecution (VEX) framework and the Java INstrumentation Engine (JINE).

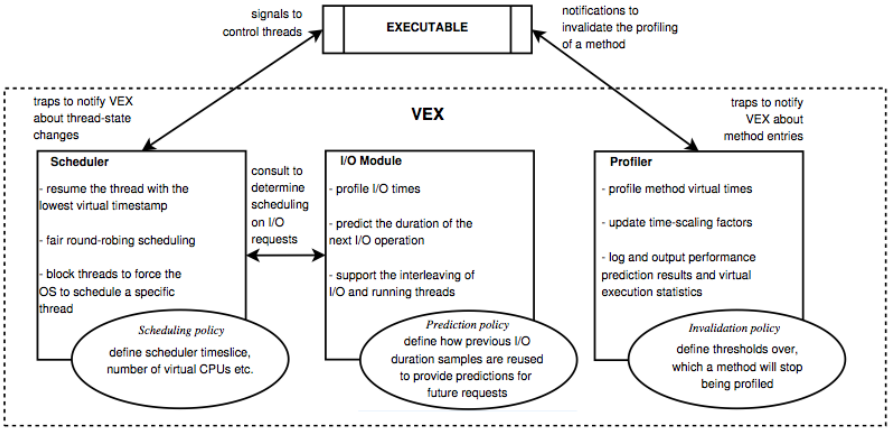


Figure 2.5: VEX design

VEX

This part is responsible for profiling methods and I/O, modelling methods and scheduling threads. The system required to do this is outlined in figure 2.5.

This system is loaded into the Java Virtual Machine (JVM) which runs the input program.

JINE

JINE acts as the interface between VEX and the Java input. JINE does this at runtime by intercepting the interesting threading and synchronisation calls made by the input program, detailed in 2.1.1. These calls are replaced/forwarded to VEX, which is executing in the JVM.

3. Design

In this chapter we will look at a possible C/C++ front end for the VTF. The resulting system will have to take an input binary and virtual time parameters to produce a virtual time profile.

3.1 Requirements

To make VTF a suitable profiler it would have to fulfill the following requirements:

Profiling
: It will have to report the exact entry and exit times of all methods to VTF.

Control
: It should be able to control the execution of the input in order to enforce VEX's virtual schedule.

Representative
: To succeed where other instrumenting profilers fall short it should minimise the observer effect. For example, the instrumentation of short methods should be avoided.

Efficient
: Any overheads caused by analysis should be reduced. Real systems are large and possibly long running, if a profiler has significant overheads it may be an infeasible option.

  Ideally it should work with minimal user preparation of the input, i.e. no source level editing or recompilation. Compilation times for C/C++ systems are in the order of hours so profilers that can instantly start sampling are generally preferred

3.2 CINE

I propose a C/C++ INstrumentation Engine (CINE) to meet these requirements. More specificially we look at in instrumentation engine built on top of the Dyninst tools [2]. These are a lightweight set of tools used parse binaries and instrument or control a process's execution through ptrace.

This approach has been chosen above the others mentioned because it is the most lightweight and flexible for instrumentation.

Compared to compiler assisted instrumentation, seen in profilers like gprof [17], the user's preparation procedure will be significantly simplified since no recompilation is needed. However, access to the compiler greatly simplifies the profiler's analysis and makes the process more reliable. Another downside of compiler assisted profiling is that instruments are fixed at compile time, ruling out the possibility of adaptive profiling, described in 2.8.

JIT 2.2.4 alternatives would also allow adaptive profiling but are significantly more heavy-weight and would have a larger observer effect. This effect may be relatively small on interpreted platforms, like Java, but the performance overhead of JITing C/C++ binaries is likely to cause a noticable overhead. On top of this, their complexity makes integration with VEX a difficult process. For example, DynamoRIO [12] does not allow for profiling code to use the *pthread* library. Since VTF is built using with a heavy dependency on *pthreads*, it would have to be re-engineered to integrate with DynamoRIO.

The hope is that the binary rewriting method will profile the input with little overhead and maintain the flexibility for adaptive profiling.

3.3 Architecture

The behaviour of CINE can be described in terms of four main components.

3.3.1 Collecter

The collecter is responsible for recording important information about the input program as it executes and notifying VEX. The collecter will take the form of a shared library which will be loaded into the input program's address space before it begins.

3.3.2 Analyser

CINE will have to be able to find important points in the input's instruction code. It could use Dyninst to parse the binary and produce a control flow graph. The graph will provide a programmatic interface for reading the binary and demangling function names. The following points will have to be found:

Entry point
: In C/C++, the *main* function is a common entry point for programs. It can safely be assumed that the user code will begin where the demangled label for main is.

Exit point
: The possible exit points for the input will be invocations of the *exit* system call or a return instruction in the body of the main method.

Thread creation
: VEX needs to be notified before a thread is created from the context of the parent and before the thread starts from the context of the child.

When a parent spawns a child it must call *pthread_create*. This can be found by searching for call instructions jumping to the address of *pthread_create*.

However, the start of the new thread is specified at run time. To find the start point, the analyser must locate the dynamic call in the *pthread* library. This requires an undertanding of the thread spawining process

Thread destruction
The termination of a thread can occur on any invocation of *pthread_exit* or if execution continues after the dynamic call to the thread's start routine

Synchronisation
Synchronisation points are found trivially by searching for call instructions to any of the known synchronisation functions. I.e. *pthread_mutex_lock*, *pthread_cond_wait* ....

Function entry
Function entry points are signalled by labels in binaries making them easy to find. Some functions may have more than one entry point, when the same name mangles to two different labels.

Function exit
Function exit points have to be found by examining the control flow graph. A function could contain a return instruction, a *(pthread)_exit* or *abort* call or any other non returning calls, [15] noreturn. The first of which to be executed would signify the end of the method.

3.3.3 Instrumenter

Once these points are identified, appropriate calls to VEX are added to the input's instructions. Care has been taken not disturb the run-time organisation of the input so that it continues to behave as expected.

Exit point
Calls to CINE should be placed here to output results to a file and gracefully exit VEX

Thread creation
Execution will have to allow *pthread_create* to create the thread. Only after the method has finished executing can the control be passed to CINE, since this is when the id of the new thread will be known.

Once the thread has spawned and been initialised, it must notify VEX that it is about to start.

Thread destruction
CINE will have to take control of exit calls to teardown its internal thread representation before the thread is destroyed

Synchronisation
CINE must capture all calls to *pthread* library synchronisation code and notify these events to VEX. These events are essential to ensure progress. If incomplete data is sent, execution is likely to lock.

Function entry
An extra call to time the function entry must be added taking care not to disturb function parameters and local variables prepared by the compiler.

Function exit
The same as the function entry, but instead preserving the return value of the function.

3.3.4 Controller

CINE will be able to control the input process at run-time using *ptrace*. One would assume that these interactions will be costly due to the inter-process communication or system call overhead. For these reasons we would like to keep these features to a minimum:

Entry point
Before the input process starts executing, VEX needs to be configured. These parameters can be loaded into the input's address space using *ptrace*.

Exit point
The controller can block on the exit system call using *ptrace*. This would allow the profiler to extract VEX's results into the its own address space for further processing.

Thread creation
An alternative to instrumenting *pthread_create* is to block on the process creation system call using *ptrace*. This would have the advantage of being able to analyse the threads state through *Dyninst* tools, this would give information on the thread's start function, stack address, underlying process id, a descriptive name... However, this would require another *ptrace* call to notify VEX of the creation of a new thread.

Thread destruction
Monitoring thread destruction with *ptrace* is not reliable since, threads may or may not be destroyed with a system call.

Invalidation
To improve the profiler's performance, a function's instruments must be removed when VEX identifies it as insignificant.

At this point CINE must communicate with the profiler which can then remove the instruments using *ptrace*. This is expected to be a costly task and will have to be executed carefully

3.3.5 Launcher

A launcher will be required to organise all these components into the final profiler. It is responsible for the configuration and the creation of the profiling environment depicted in 3.1. It does this through the following process:

1. Parse the VEX arguments
2. Spawn the input process and halt execution
3. Load VEX and CINE libraries into the input's address space
4. Trigger analysis on the input binary
5. Trigger instrumentation on the input binary
6. Initialise the VEX and CINE libraries
7. Register the controller to manage the input process
8. Resume the input process
9. Collect results on termination

3.4 Integration with VEX

In order for CINE to successfully integrate with VEX, the collector must communicate with VEX at all the instrumentation points listed.

The most efficient way to do this is to load the VEX library into the input program's address space. Each thread will invoke VEX directly at the instrumentation points. VEX's code will then be able to pause threads as required. VEX will need to have it's own thread of execution that is treated separately from the input program's threads so that it can schedule the other threads.

3.5 The complete system

The resulting system is illustrated in figure 3.1. The user will run CINE with the input code path and any input code arguments. CINE will use the BPatch API, and any sub APIs if required, to analyse and instrument the input code. To run the input code, CINE will inject the Dyninst run-time, CINE collector and VEX shared libraries into the input's address space.

The Dyninst run-time will contain trampoline code for the instruments. These trampolines will manage the input state before jumping to CINE collector code. This library will also be responsible for communication with the Profiling process.

The CINE collector will act as an interface between the input code and VEX. It will collect relevant information from the input code's state and make the corresponding calls to VEX.

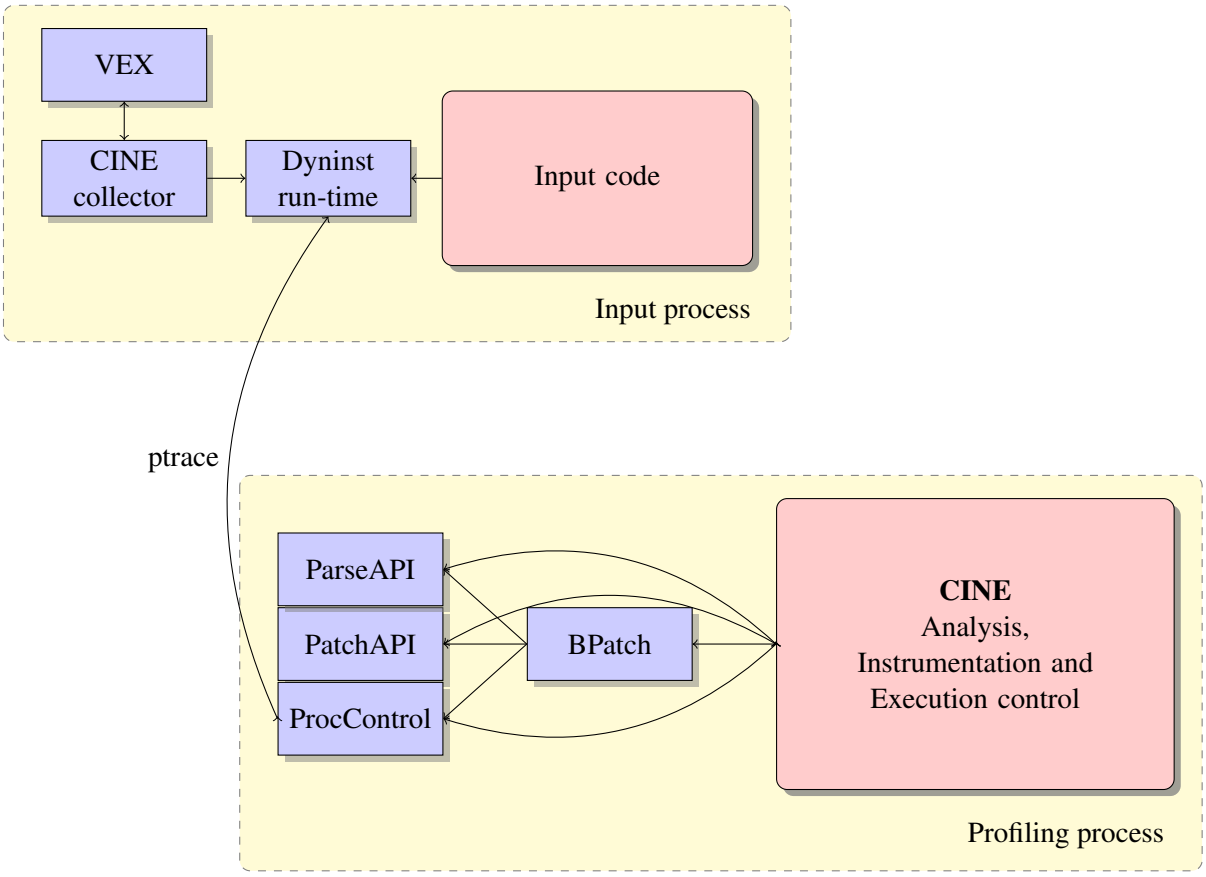


Figure 3.1: The complete CINE system

4. Implementation

4.1 Implemented system

CINE consists of two implementations of the design outlined in figure 3.1.

One is a static instrumentation framework, CINE static. It rewrites the input binary to load the CINE collector into its address space. It then rewrites function entry and exit points to call the collector. The edited binary is saved written to storage so the user may run the input as desired. This lightweight framework trades flexibility for performance. The hope is that by manipulating the input's execution as little as possible, the run time overheads will be small.

The second is a dynamic instrumentation front-end, CINE dynamic. This framework instruments the binary in the same way the former does. However, it then launches the binary and attaches to it using *ptrace*. CINE will then remove superfluous instruments as the input executes. This alternative hopes to reduce overheads by performing extra run time analysis on the input.

4.2 Challenges

This chapter will touch on some of the challenges and solutions that were discovered to realise the current system.

4.3 Analyser

4.3.1 Function bounds

One responsibility of the analyser is to find the function entry and exit points in a binary. At first it would make use of the BPatch API to do this. The BPatch abstraction for functions, *BPatch_function*, includes the following member function:

```
enum BPatch_procedureLocation {
BPatch_entry,
BPatch_exit,
BPatch_subroutine,
BPatch_locInstruction,
```

```
BPatch_locBasicBlockEntry,
BPatch_locLoopEntry,
BPatch_locLoopExit,
BPatch_locLoopStartIter,
BPatch_locLoopStartExit,
BPatch_allLocations }

const std::vector<BPatch_point *> *BPatch_function::findPoint(const
  BPatch_procedureLocation loc)
```

Although this may appear adequate to find these points, it has turned out to be unreliable. In particular Dyninst's approach for finding the function exit is prone to error. From the entry point Dyninst will parse the function's body until it finds a return instruction. Let's say an input binary makes a call to a procedure that does not return, like *pthread_exit*, *exit*, *abort* or any custom method annotated with GCC's "noreturn" attribute. On X86, GCC compiles noreturn calls to look identical to returning calls and instead omits the return statement at the end of the function. So the result of instrumenting these functions with Dyninst is that instruments are added at the end of an arbitrary function later on in the binary.

Below is an illustration of the final timing instrumentation method. Below is the disassembly of the *inc_count* function from the example shown in A. From examining the source, it is obvious that the function end point is at $400bd2$. On instrumenting this method with the BPatch API, instruments will be placed at address $400e70$.

```
0000000000400b16 <inc_count>:
  400b16: 55                    push   %rbp
  400b17: 48 89 e5              mov    %rsp,%rbp
  400b1a: 48 83 ec 20           sub    $0x20,%rsp
  400b1e: 48 89 7d e8           mov    %rdi,-0x18(%rbp)

 ...

  400bc3: 83 7d fc 09           cmpl   $0x9,-0x4(%rbp)
  400bc7: 0f 8e 69 ff ff ff     jle    400b36 <inc_count+0x20>
  400bcd: bf 00 00 00 00        mov    $0x0,%edi
  400bd2: e8 d9 fd ff ff        callq  4009b0 <pthread_exit@plt>

0000000000400bd7 <watch_count>:
 ...

0000000000400e10 <__libc_csu_init>:
 ...
  400e70: 41 5e                 pop    %r14
  400e72: 41 5f                 pop    %r15
  400e74: c3                    retq
  400e75: 66 66 2e 0f 1f 84 00  data32 nopw %cs:0x0(%rax,%rax,1)
```

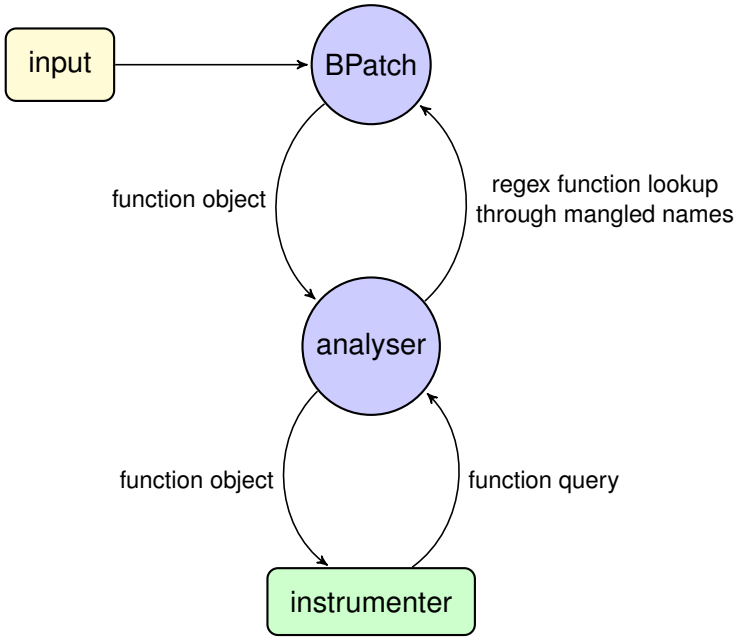
```
  400e7c: 00 00 00 00
```

CINE will parse the function body for noreturn functions and stop false instruments from being placed. The below example shows the outcome of this parsing. The function end point is instrumented at 400*bcc* and parsing stops for this method.

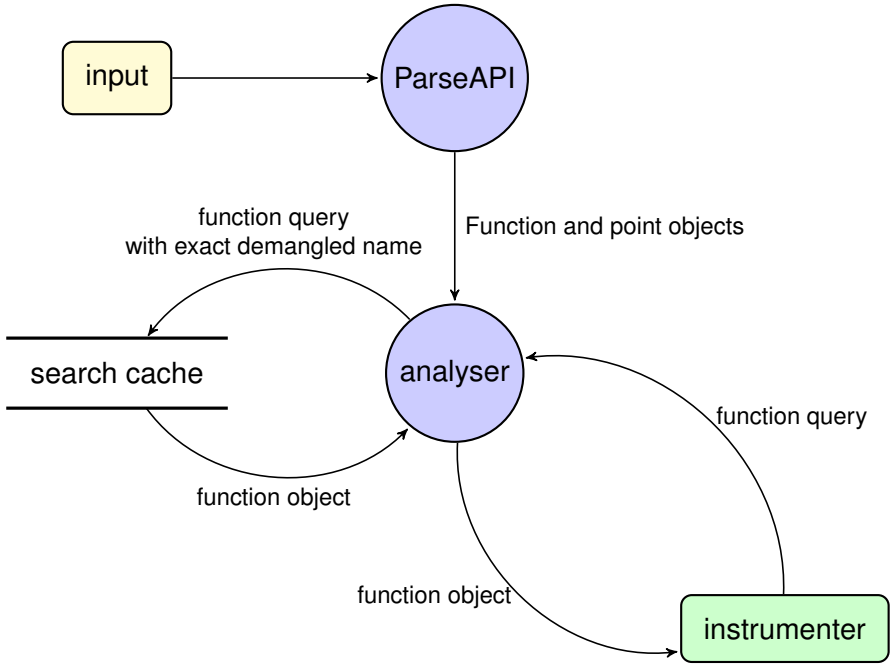
```
0000000000400b16 <inc_count>:
  400b16: e9 4c 0b 30 00        jmpq   701667 <inc_count_dyninst>
  400b1b: 83 ec 20              sub    $0x20,%esp
  400b1e: 48 89 7d e8           mov    %rdi,-0x18(%rbp)

...
  400bcc: ff e9                 ljmpq  *<internal disassembler error>
  400bce: b4 0b                 mov    $0xb,%ah
  400bd0: 30 00                 xor    %al,(%rax)
  400bd2: e8 d9 fd ff ff        callq  4009b0 <targ4009b0>
...
```

4.3.2 Parsing optimisation

The analysis of medium sized programs using the BPatch API was taking a very long time. For analysis, CINE must find all the functions it wants to replace and all the calls to these functions. The naive implementation was to look up each function and then parse the binary for calls to that function. Profiling CINE with OProfile revealed that much time was spent in Dyninst performing regex searches for each function lookup and parsing function bodies for function call expressions. However, the sampling method meant that the call graphs were incomplete and shallow. This mean that the exact cause of the inefficiencies could not be identified.



Steps were taken to optimise the analysis code. CINE now manually parses the function bodies to find and instrument all relevant call points in one pass. Caches have been set up to reduce the number of expensive regex searches that were performed in Dyninst.



The new parsing times were still unsatisfactory[1]. This may be acceptable for long running programs where the preparation time is not a factor. A comparison of parsing times and compilation times can be found in the evaluation.

Further improvements to the parsing stage would involve parallelisation or an implementation of partial system profiling, suggested by [4]. With the latter method, overheads and preparation time are reduced by only instrumenting the part of the system that is most interesting to the performance engineer.

4.4 Controlling Pthreads

CINE has to notify VEX of all synchronisation calls, thread declarations and as soon as threads begin to execute, as explained in section 3.3.3. Various methods for controlling *pthreads* have been tried, with varying levels of success.

4.4.1 Call instrumentation

The simplest solution is to find all the call points in the input that jump to the Pthread library and rewrite them to jump to the CINE library. This was trivial for most of the points mentioned in section 2.1.1.

However this method should have high overheads and it does not scale well for larger code bases. The whole binary has to be parsed for calls to the *pthread* library code. One would expect this method of rewriting to be relatively cheap since we only intend to manipulate the call address. However, the frequency of jumps to trampoline code can be seen in the example instrumentation of *inc_count* in appendix C. Notice the jumps to "dyninst_inc_count" before calls to functions that were before prefixed with "pthread".

4.4.2 Interposing

Interposing is the process of loading an alternate version of a shared library into the input's process. The input will then call the phony library as if it were the real one [10]. In our case, a phony *pthread* library will be created to make calls to the CINE collector. The CINE controller will then have to replace the real *pthread* library.

This method appears to be very efficient in terms of analysis. The *pthread* functions mentioned in section 2.1.1 can be identified and rewritten to call respective functions in the CINE library. The core input code will then remain unchanged. In this case the overheads would scale very well. As the input code gets larger and more calls to the *pthread* library are made, the thread control cost will remain constant.

[1] Perhaps I could have foreseen this by making CINE profile itself in virtual time, which may have been viable if the Dyninst libraries could be parsed in a feasable time.

```
1  //input binary
2    ...
3    //this is a call to the phony library implemented by CINE
4    callq 0x___ <pthread_mutex_lock>
5    ...
6
7  //phony libpthread
8  pthread_mutex_lock:
9    0x___ ...
10   //this is a call to give control of the thread to VEX
11   callq 0x___ <ThreadEventsBehaviour::onRequestingLock>
12
13 //libvex
14 ThreadEventsBehaviour::onRequestingLock:
15   ...
16   //this call is meant to block the thread
17   //but calls the phoney libpthread
18   callq 0x___ <pthread_mutex_lock>
```

Figure 4.1: Instrumentation that causes an infinite recursion between *pthread_mutex_lock* and *ThreadEventsBehaviour::onRequestingLock*

Unfortunately, VEX is linked against the same *pthread* library that the input code will be using. VEX relies on *pthread* to create a scheduling thread and synchronise itself. Naive implementations of this method result in infinite recursion or deadlocking for affected *pthread* calls.

An example of this infinite recursion is shown in 4.1. The input code will reach line 4 and jump to the CINE's phony *pthread* library at line 8. CINE will collect the relevant data and jump to VEX on line 14. VEX will want to pause the thread to enforce its own schedule and will do so by calling the *pthread* library. However, this will jump to the phony library and cause a loop.

However, CINE has not been able to solve this problem. Attempts were made to statically link VEX, and its dependent libraries, to an alternate *pthread* location. This could potentially be done by using the dynamic loader interface [13]. The dynamic loader interface provides functions like "dlopen" and "dlsym" that allow you to reference symbols from other libraries programmatically. This approach caused several symbol look-up errors at run-time. Despite this, there is scope for development. To complete this implementation, the VEX code base and its dependent libraries must be investigated so that they can be completely and reliably re-linked to an alternate location, as shown in figure 4.2

4.4.3 Instrumenting the start of a thread

Finding a safe point to instrument the beginning of a new thread required some care. New threads must pass control to VEX before executing. When Pthreads are created, they are dynamically

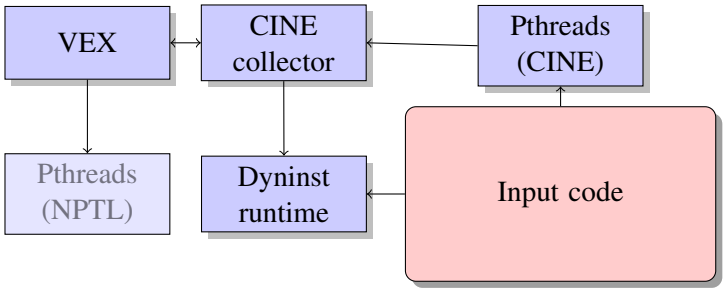
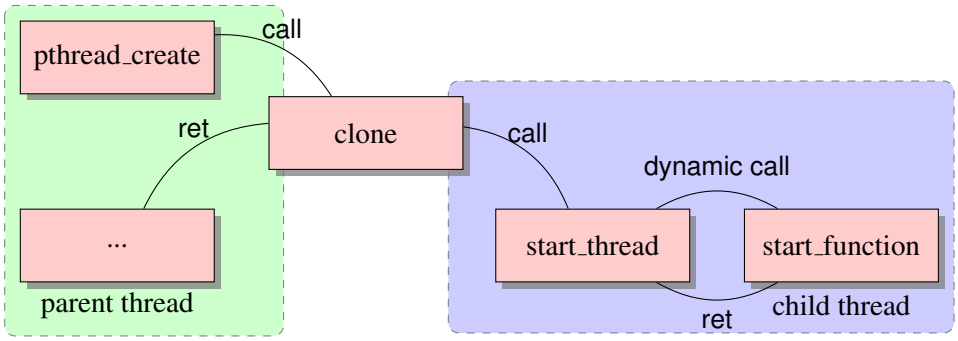
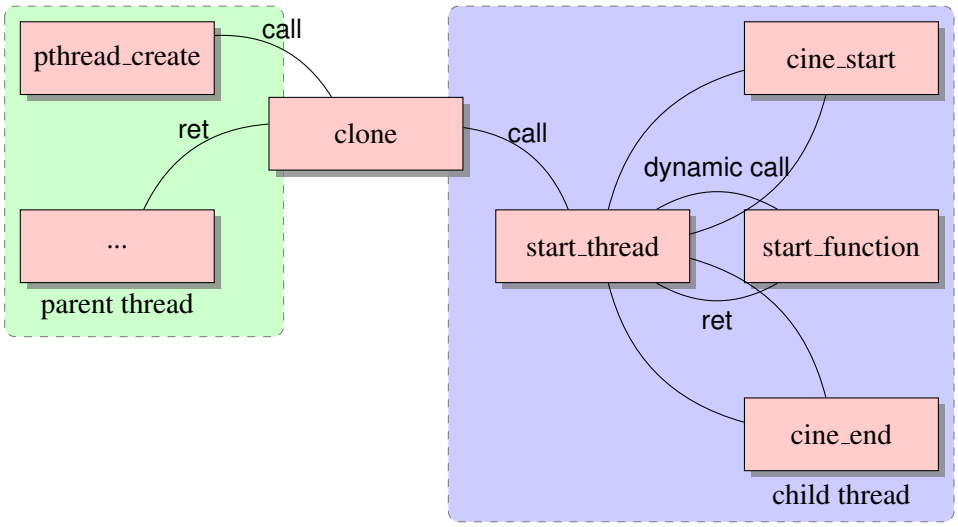


Figure 4.2: A run-time analysis system that relocates *pthreads*

launched at a start routine specified by the input code at run-time.



This means instruments have to be added to the *pthread* library's thread initialisation code to catch new threads. The position of these calls is important, they have to be late enough in the initialisation process so that the thread can safely execute code in VEX. CINE's is able to reliably catch the start and termination of threads after trialing various instrumentation points. The current implementation parses the *pthread* library's *start_thread* function body for its sole dynamic call instruction and inserts instruments to call CINE before and afterwards.



Ptrace

An attempt was made to use *ptrace* to notify the profiler on system calls from the input to create and exit threads. This would have the advantage of being less tightly coupled to the *pthread* library than the previous approach. This would be done in the profiler's process space by blocking on the "system call" *ptrace* request. At this point CINE could analyse the input for information on the new thread. CINE will then have to notify the VEX, running in the input's address space of this information.

The Dyninst tools limit the effectiveness of this method. Although undocumented, the tools do not allow the profiler to execute code on the input upon being notified of a system call. This makes it difficult to notify VEX thread creation. On further investigation of the Dyninst source and the developer mail archives [22] it has been found to be a non-trivial fix. CINE works around this by inserting instruments at the first available point of the thread's execution and continues the input. However, the frequent inter process communication produces large overheads for programs that create threads often so this has been left as an optional feature.

Using *ptrace* for thread or process termination has proven to be unreliable. It appears that *ptrace* can notify the profiler on explicit calls to *pthread_exit* or the *exit* system call. This means that when execution ends on return from *main* or a thread's start routine, the profiler will miss the exit and will fail to notify VEX. CINE implements a variety of methods so that all events are caught. As a fall back on using *ptrace* to monitor for the input's termination, instruments are put in the ELF loading code, in *_start*, that should be executed on return from *main* [28].

4.5 Invalidation

CINE removes instruments from methods that are flagged by VEX's invalidation policy to reduce the execution overheads.

This is implemented with Dyninst's user message interface. This interface allows the input program to communicate with the profiling process asynchronously. To pass messages, the input stores the data at a known location in the Dyninst run-time library and breaks the input process with a Dyninst breakpoint. Linux then gives control to the CINE controller since it is registered as the signal handler. CINE uses *ptrace* to read this data from the input's address space and continue the process. The data is used to find the invalidated method and remove its instruments.

This method has a number of costs. Signal handling takes a long time to manage and should be avoided to minimise overheads. There is also delay in the removal of instruments. This becomes an issue in shorter running programs where the reaction of the profiler is too slow to have an impact on the profile.

For functions that are executed frequently, this method results in re-invalidation. This is when VEX will be asked to invalidate functions several times, or after it has already been invalidated. Care had to be taken to deal with these cases gracefully. To do this, VEX is modified to immediately remove tracking data on invalidated functions and discard any invalidation of untracked functions. This approach sacrifices VEX's information on the actual instrumentation state of the input so that less communication is needed between the input process and the profiling process.

CINE does have an incomplete implementation of a fast synchronised communication channel between the input process and the the profiler. A named pipe has been set up to allow multiple threads notify the profiler on invalidation. The profiler would then have to use the pipe to signal the thread to continue. The difficulties with implementing this system will be with the high overhead of inter-process communication and managing concurrent invalidations.

For example if a thread sends an invalidation message, a context switch occurs and another thread attempts to invalidate the same method the profiler will have to wake up both threads when the instruments have been removed. A named pipe alone is not powerful enough to manage synchronisation cases like this.

4.6 Trampoline recursion protection

Trampoline recursion protection is a system put in place by Dyninst to prevent infinite recursion caused by instrumenting the code. An example of this can be seen with the *pthread* separation problem outlined in section 4.4.2.

To prevent this, Dyninst drops any snippets that are executing at the same time. Even with careful instrumentation, this method results in too many false positives and drops snippets unecessarily. By not executing snippet code, VEX would receive incomplete information and cause deadlocks. For this reason trampoline recursion protection must disabled in CINE.

4.7 VTF

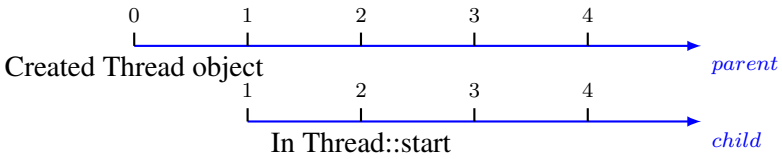
VTF was designed to test virtual execution in Java. A few semantic changes were required to correctly integrate VEX with C/C++.

4.7.1 Thread creation

Java thread creation is more verbose than *pthread* creation. In Java a programmer makes a thread object that can be configured and started programmatically.

```
//some form of declaration of a "Thread" object
//or object that implements "Runnable"
Thread t = new Thread(runnableObject, "exampleThread");

//the start point for the thread.
t.start();
```

What JINE is able to do in this case is instrument a the creation of a thread object and the explicit start of a new thread.



Pthreads are created with a single call to the *pthread* library. Threads can begin to execute as soon as they exist in the system.

```
//creation of thread
pthread_t thread;
pthread_attr_t attr;
char *arg = "example argument";
pthread_create(&thread, &attr, function_pointer, (void *)arg);

//only at this point is the "thread" variable initialised
//with all the data VEX requires.
```

To notify VEX of thread creation, CINE must instrument the return of the *pthread_create* method to obtain the child thread's handle. As mentioned in section 4.4.2, the *start_thread* method is then instrumented to signal the start of a thread's execution.

The problem with this method is that the Linux scheduler could start the new thread before the parent thread has notified VEX causing it to fail, as in figure 4.3.

CINE prevents this from happening by enforcing a safe schedule. A set of registered threads is maintained, they are made thread safe by conditional locks that will notify waiting, premature

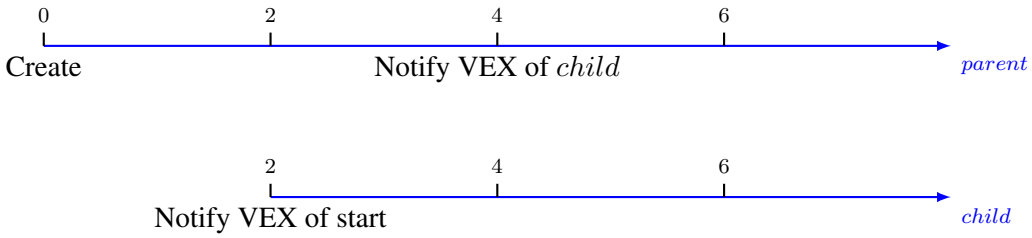


Figure 4.3: CINE prematurely notifying VEX of the start of a thread

threads on new registrations to VEX. Cases like this are handled as shown in figure 4.4

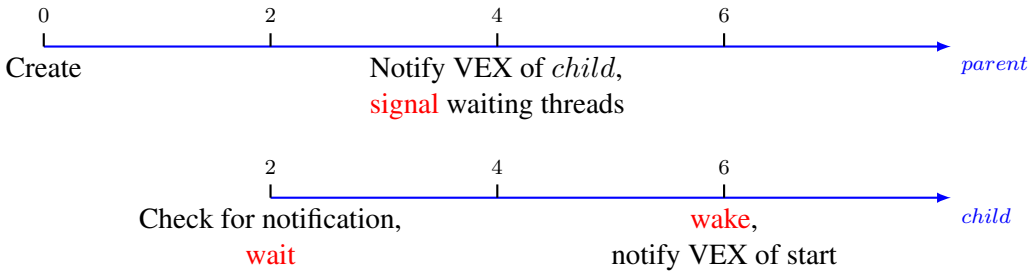


Figure 4.4: Final implementation that synchronises the creation of threads

4.7.2 Conditional wait

Java monitors are a simpler version of conditional waits. In Java, monitors lock and notify the same object.

```
synchronized(SensitiveObject) {
  //sensitive code
  //...

  //releases the lock on SensitiveObject and sleeps
  wait();
  //...

  //notifies threads waiting on SensitiveObject
  notify();

}
```

The existing VTF API for JINE reflects this:

```
bool beforeReleasingLockOfAnyReplacedWaiting(const long &objectId, const bool &
    updateLockRegistry);
bool onReplacedWaiting(const long &objectId, const
  bool &updateLockRegistry);
```

These events were created so that VEX could react appropriately to a thread entering a monitor, explained in [4].

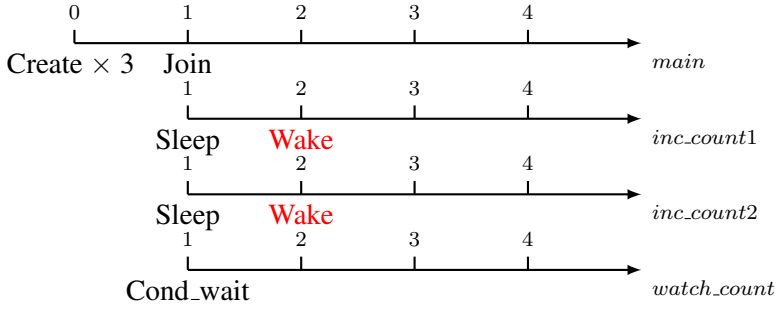
A simple extension was required to make VEX behave correctly for *pthread* conditional waits. The API had to be extended to allow a condition and a mutex to be specified. The new implementation allowed for threads to contend for locks that were separate to the condition when performing a conditional wait, for example appendix A line 36. The extended API looks like this:

```
bool beforeReleasingLockOfAnyReplacedWaiting(const long &objectId, const bool &
    updateLockRegistry);
bool beforeReleasingLockOfAnyReplacedWaiting(const long &cond, const long &
    mutex, const bool &updateLockRegistry);
bool onReplacedWaiting(const long &objectId, const bool &updateLockRegistry);
bool onReplacedWaiting(const long &cond, const long &mutex, const bool &
    updateLockRegistry);
```

4.7.3 Forward leaps

Virtual Forward Leaps (VFL) are jumps forward in virtual time. These occur in VEX to save time in execution when it is known that the system state will be unchanged for a certain period of time so the next thread can be scheduled early. As discussed in [4] this is an experimental feature that requires further investigation. To infer whether a forward leap can be made several heuristics are made. Unfortunately, these heuristics can produce false negatives that can lead to loss of liveness in with particular test programs.

One example is with multiple sleeping threads. For the mutexthreads example, appendix A, once the *watch_count* thread has gone to sleep, both *inc_count* threads have gone to sleep and the main thread is waiting to join, VEX is in the following state:



At time step 1 VEX should jump to the time that the first *inc_count* thread will wake up and continue its execution. However, the heuristics would falsely conclude that execution could continue.

To see why we can examine the heuristics one by one:

1. If a thread's time slot expires before the global virtual time it must always be allowed to VFL.

   This does not apply to our case because all of the threads are executing at the same rate of virtual time, i.e without speedup, so no thread will be behind the global virtual time.

2. A thread, t, must not VFL if for all other threads that will expire before t, T_L, there is a $t' \in T_L$ where t' is:

   (a) Suspended or running

   (b) Performing I/O

   (c) Natively waiting

   This blocks the VFL at time step 1 because the *main* thread is natively waiting to Join.

3. No VFLs should be allowed if there is a change in the global virtual time between time slices.

   With all the threads either inactive or waiting to join, there will be no change in global virtual time.

4. If the system state has not changed between time slices and all threads are stopped, sleeping or uninterruptible sleep then VFLs should be allowed.

   This heuristic should resume either of the *inc_count* threads at time step 2 after noticing that all the other threads are inactive. However, VEX's implementation would only class threads in the "Suspended" state as inactive and discounts threads that it holds in a "Timed-Waiting" state, i.e the other *inc_count* thread.

A simple extension was added to allow VFLs for multiple sleeping threads. This patch is unlikely to make a difference in intensive benchmarks where sleeping does not often happen. However, asynchronous systems that often have multiple sleeping threads would not work with CINE otherwise.

5. Evaluation

In this chapter we will compare CINE to other popular profilers to evaluate how effective it is as a performance engineering tool.

The PARSEC benchmark suite is being used to evaluate the system, [5]. It is geared towards generating a variety of parallel workloads so is likely to reveal much insight on CINE's handling of concurrent programs.

Blackscholes A financial calculation benchmark with few, long running functions. There is also very little inter-thread communication. This is compatible with all the systems produced

Facesim A simulator for facial movements. Like the blackscholes benchmark, the parallelism is very coarse grained. However, the work load is the largest and there are more function calls than any of the other benchmarks.

Canneal A processor design optimiser. The algorithm used in this benchmark has a fine grained parallelism and uses atomic instructions to synchronise. This will give an idea of how CINE manages with code that does not synchronise through VEX

Raytrace A visualisation benchmark that dynamically balances the workload between threads. This should reveal how CINE reacts to less predictable threads

The performance on these benchmarks will be compared to those of popular profilers so that the techniques used can be compared. We will look specifically at *gprof* and *oprofile* in order to draw a comparison between the optimised instrumenting/tracing profilers and our hybrid alternative.

5.0.4 Unused benchmarks

Some benchmarks are not yet compatible with CINE/VEX and therefore have been executed on a single thread or dismissed altogether. The most intensive benchmarks leave CINE vulnerable to a variety of faults:

1. VEX will deadlock if multiple threads try to access the PAPI performance counters at the same time. This occurs when several threads are started at the same time, VEX will attempt to stop them and will lock in *_linux_get_virt_nsec_gettime*, which is not thread safe.

2. Dyninst is prone to killing executions that intensively invalidate functions with a "Bus error".

3. CINE/VEX misses a synchronisation call. In this report we are focusing on the subset of synchronisation calls outlined in section 2.1.1

5.1 Overheads

The extra time taken to profile an application should scale well for inputs. Performance engineers are most likely to be used with complicated, intensive systems so it is important that the CINE will be able to manage these inputs too.

At first, CINE incurred very large overheads for the majority of benchmarks, figure 5.1.

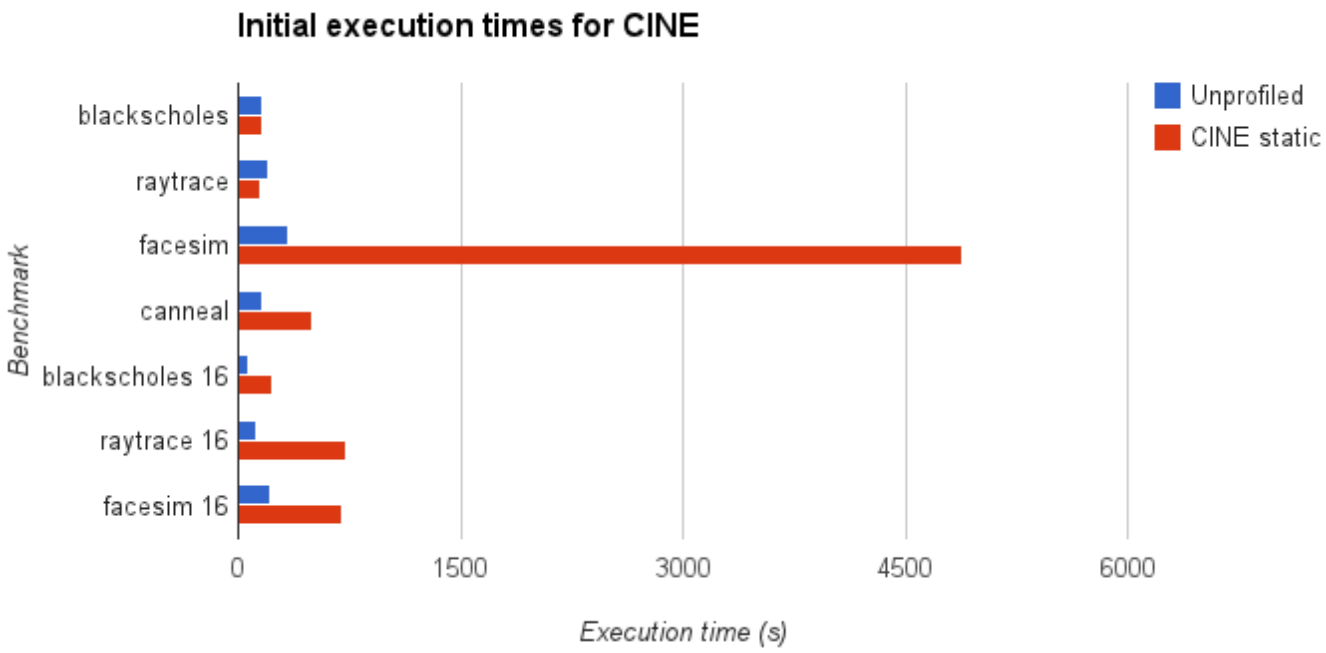


Figure 5.1: Initial profiling times of the PARSEC benchmarks. Overheads ranged from 0.7% on Blackscholes to 1367% for faceism. Notice that when the benchmarks are run on 16 threads there is a significantly larger overhead on execution. This is likely due to the extra synchronisation added by CINE to conform to VEX's thread model, as mentioned in 4.7.1

On closer inspection one would notice that CINE instruments 12 methods for blackscholes and 3736 methods for facesim. Notice that benchmarks like blackscholes have a very low overhead where as facesim takes over 14 times longer to execute than the unprofiled execution time.

Binary instrumentation could be a cause of this. As mentioned in the Dyninst documentation, instrumentation is costly and should be only inserted when necessary. *Gprof* manages to reduce its

instrumentation cost by leveraging the compiler, it is able to optimise calls to *gprof*'s counter code whereas elaborate trampoline code produced by binary rewriters can significantly slow execution. JIT instrumenters theoretically are able to relocate and change code without restriction to similarly optimise instrumentation.

Adaptive profiling

One solution to reduce these overheads is is adaptive profiling. The problem with this solution is that there are numerous policies to chose from. We examine the effect of these policies on two different programs in, figures 5.2, 5.3 and 5.4.

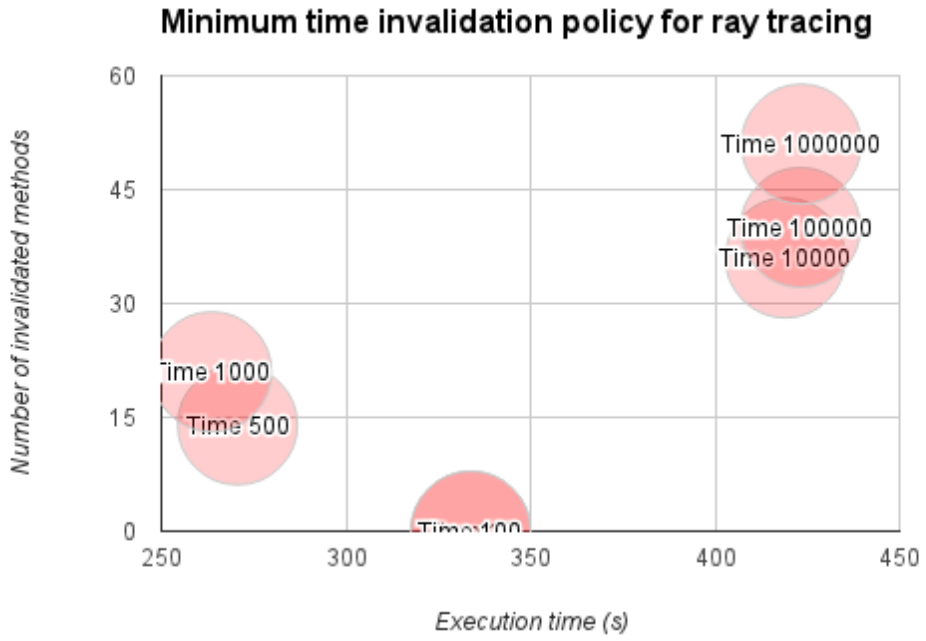


Figure 5.2: Varying the minimum time invalidation policy while profiling the raytrace benchmark. Each policy "Time x" invalidates functions that are shorter than x milliseconds. The cluster in the top right represent the aggressive policies that invalidate too many methods. The performance penalty of the excessive invalidations can be seen by how the execution time is worse than for "Time 100", which invalidated no methods. The cluster on the left show the optimal policies in terms of overheads. Execution sped up by 18% with the optimal time invalidation.

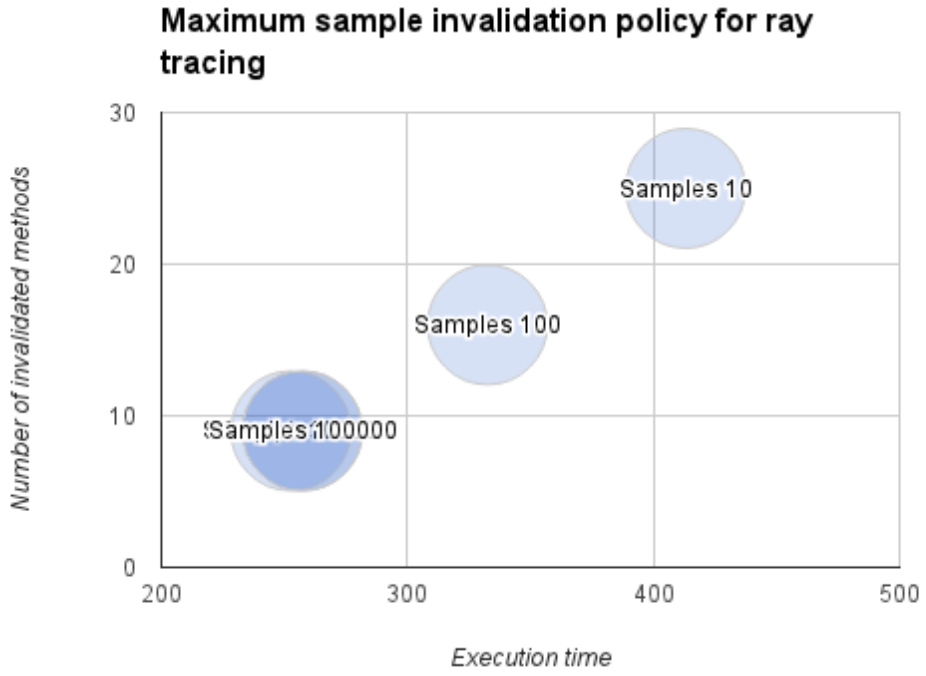


Figure 5.3: The ”Samples x” policies invalidate functions that are called over x times. In the same way as the ”Time” policies, aggressive policies suffer the ”over-invalidation penalty”. The number of samples is a more deterministic measure and will generally produce consistent results that are related to the input code algorithmically. For example in ray tracing, Samples 10000+ invalidates mostly the pair constructor and vector insertion methods. This cuts down the overheads substantially and only 9 functions are invalidated. At samples 100, vertex traversal functions are removed, only 7 more functions are invalidated but a 29% slow-down is suffered.

5.1.1 Run-time overheads

Run-time overheads are those that increase the running time of the input. These times should be kept to a minimum so that long running, complex programs can be profiled in a feasible time. The execution times in figure 5.5 were calculated with the invalidation policy set to ”Samples 1000”, the best performing policy.

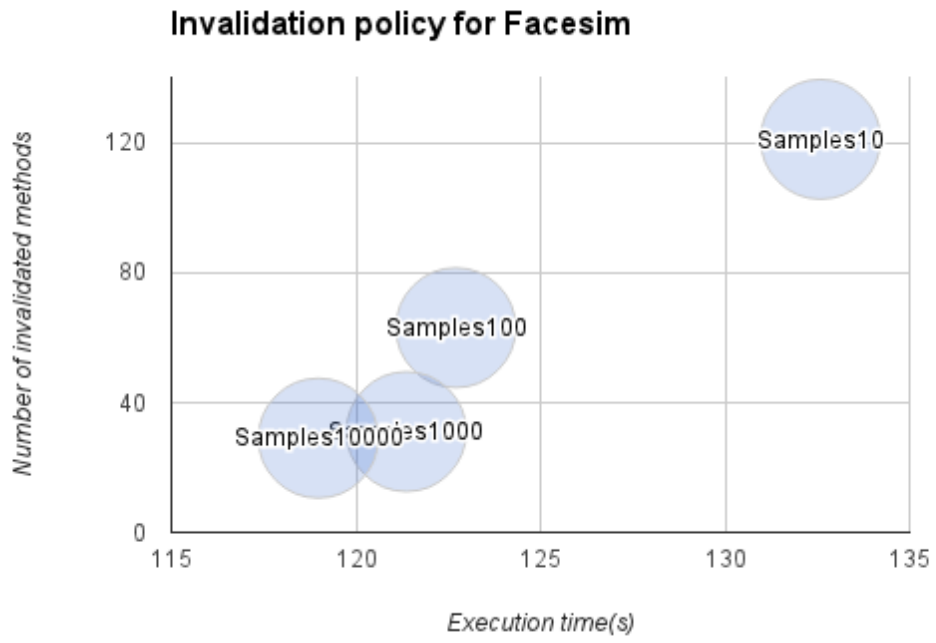


Figure 5.4: The "Samples x" policies applied to Facesim. Although the variance in improvement of the policies is less than in the raytracing benchmark, invalidation is necessary to make this profile feasible, the execution speedup is 95%! Surprisingly the minimum time policies had no effect on the facesim benchmark. Deeper inspection reveals that the helper functions in face simulation generally iterate through arrays of data, thus taking long enough to evade the invalidation policy.

5.1.2 Preparation overheads

Instrumenting profilers normally require preparation time. In this time the input code is analysed and instruments are inserted. For example, *gprof* will require recompilation and CINE will have to parse the input binary. The relationship between benchmark size and instrumentation size is shown in figure 5.6

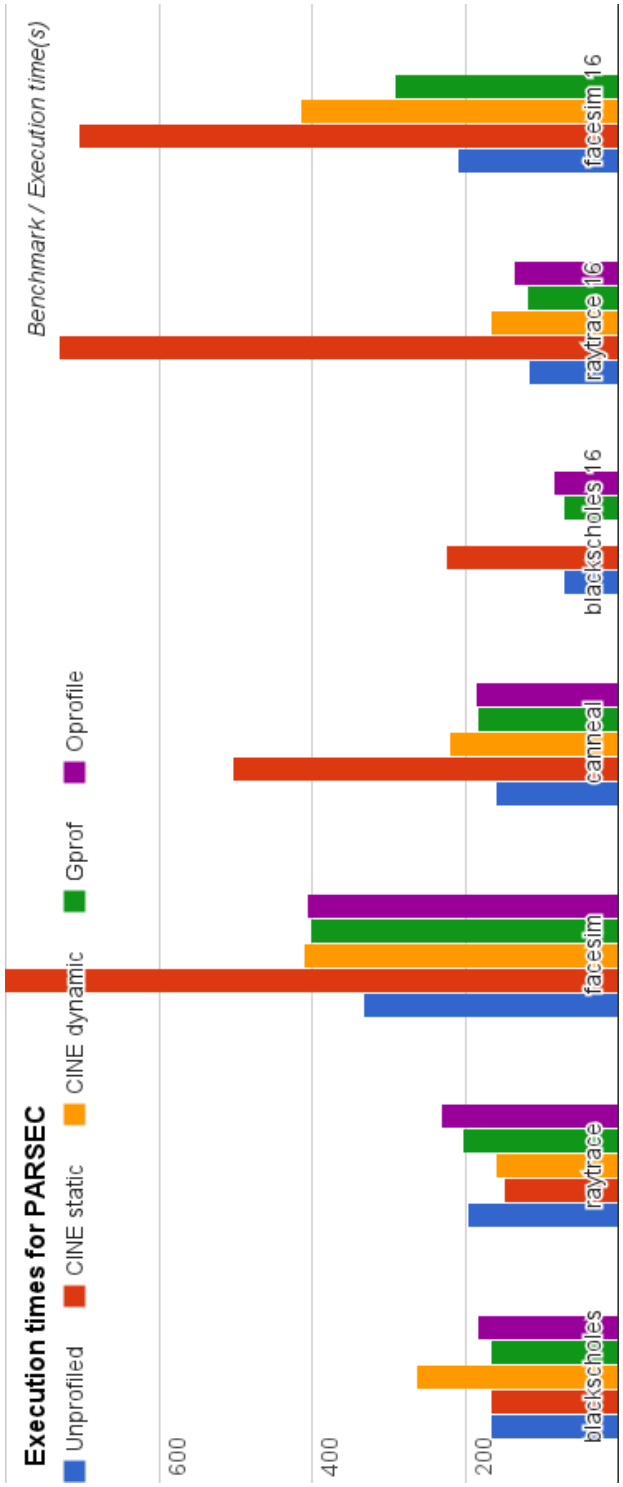


Figure 5.5: The execution times for the benchmarks for all the profilers in this comparison.

By circumventing *pthreads* synchronisation, CINE is able to beat *gprof* and *oprofile* for the simpler benchmarks. In blackscholes and raytrace, VEX will only have one thread on its virtual timeline and therefore will essentially ignore any synchronisation events.

However, on the multi-threaded benchmarks the performance of CINE appears to deteriorate as threads require more communication.

CINE dynamic manages to maintain an overhead that is close to the other profilers for the majority of the benchmarks. It has a lowest of 19% for facesim and a highest of 49% in facesim 16.

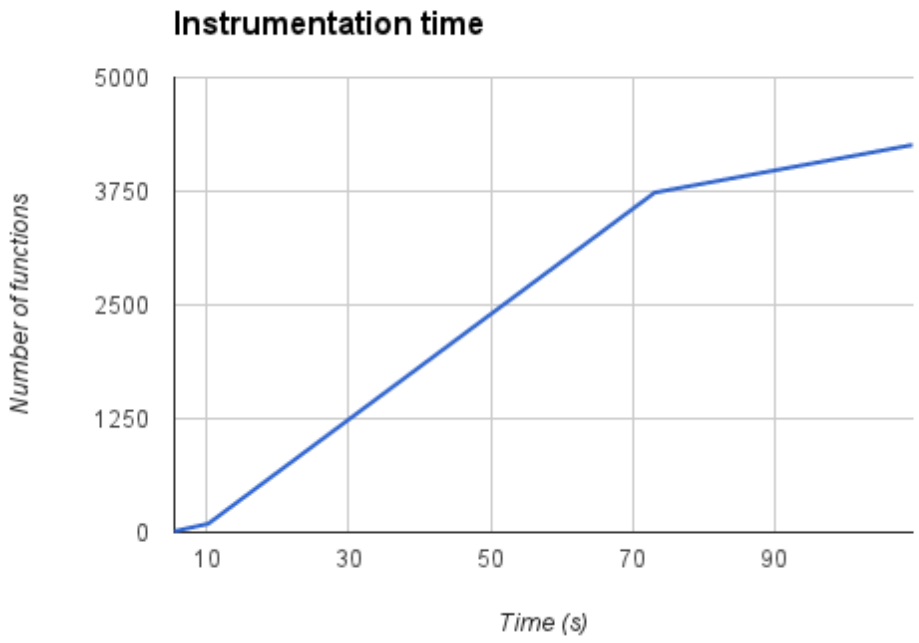


Figure 5.6: The linear relationship between instrumentation time and the number of functions in the benchmarks used. As code bases become large, instrumentation will become infeasible. The interposing approach suggested in section 4.4.2 could make instrumentation a constant time process. Paired with techniques like partial profiling, suggested in [4], a fraction of the binary could be analysed reducing both instrumentation run-time and preparation overheads.

5.2 Accuracy

A comparison of the profiles produced by each profiler is shown in table 5.1. It shows the average time spent in each of the top commonly profiled functions. Although CINE static suffers severe overheads because of lack of invalidation, its complete profile is more consistent with those of GProf and OProf. CINE dynamic has a tendency to have higher CPU times for higher level functions because of its invalidation policy.

5.2.1 Predictive accuracy

The main goal of CINE is to act as a binary front end for VEX. We therefore have to test that it has integrated correctly. A simple experiment was set up to test that VEX is able to predict the execution time of a trivial program, appendix D. The results are shown in figure 5.7.

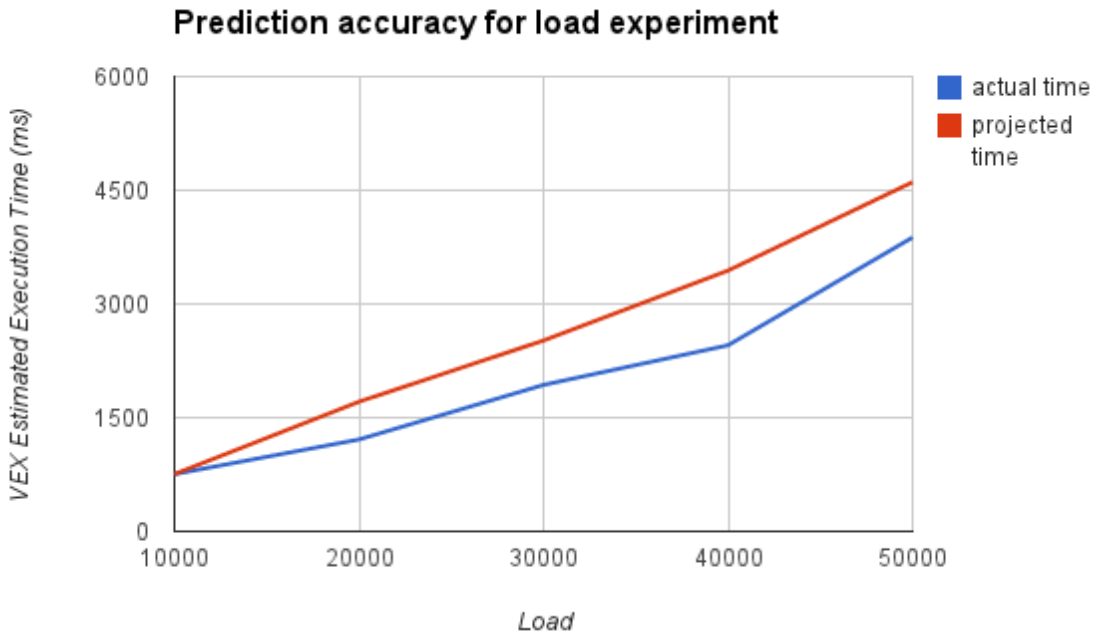


Figure 5.7: A plot of the prediction test results. The blue series represents the execution time of the test program with loads ranging from 10000 to 50000. The red series was estimated by executing the test program under VEX and slowing down the ”RunningThread” function by a factor of 1-5.

	CINE dynamic	CINE static	GProf	OProf
blackscholes				
bs_thread		88.49	83.81	28.4334
main		11.15		
CDF			16.43	
blackscholes 16				
bs_thread	91.33	90.37	80.23	28.9044
main	8.67	9.63		
CDF			20.02	
facesim				
PhysBAM::DIAGONALIZED_F...	21.02			16.3023
PhysBAM::MATRIX_3X3<float>::operator*	invalidated			9.6542
DIAGONALIZED_FACE_3D<float>::dP_From_dF	19.39			3.3694
PhysBAM::DIAGONALIZED_I...	10.3			1.1588
taskQIdleLoop				0
PhysBAM::DIAGONALIZED_FINITE_V...				0.6393
facesim 16				
PhysBAM::DIAGONALIZED_F...	69.37	31.57		18.0742
PhysBAM::MATRIX_3X3<float>::operator*	invalidated	18.41		10.7802
DIAGONALIZED_FACE_3D<float>::dP_From_dF		12.28		3.1417
PhysBAM::DIAGONALIZED_I...		6.6		1.2937
taskQIdleLoop	23.17			0.0113
PhysBAM::DIAGONALIZED_FINITE_V...	4.41			0.7288
raytrace				
ObjsParser::Parse	54.11	41.44	6.27	0
Context::renderFrame	44.77	12.92	8.08	6.5442
RTTL::BinnedAllDimsSaveSpace::recursiveBuildFast	0.27	10.62	4.31	4.5362
RTTL::StandardTriangleMesh::storePrimitiveAABBs	0.25	0.07		0
RTTL::TraverseBVH_with_StandardMesh	invalidated	12.48	42.46	55.9252
raytrace 16				
ObjsParser::Parse	98.45	62.55	12.51	0.3263
Context::renderFrame	0.01	0.07	0	0
RTTL::BinnedAllDimsSaveSpace::recursiveBuildFast	0.52	15.77	0	3.4668
RTTL::StandardTriangleMesh::storePrimitiveAABBs	0.27	0.11		0
RTTL::TraverseBVH_with_StandardMesh			34.47	67.2092

canneal				
annealer_thread::Run	80.43	22.73	8.13	7.7126
netlist::netlist	16.09	3.07	1.58	1.1687
netlist::total_routing_cost	1.63	0.36	0.2	0.1898
netlist_elem::swap_cost		45.88	76.58	76.1168

Table 5.1: Comparison of profiles

6. Conclusion

CINE is indeed a Virtual Time Profiler. However, there are points to note from the evaluation.

6.1 Instrumentation

CINE is able to instrument all standard functions in a binary and control threads by redirecting calls from the *pthread* library. This redirection technique, explained in 4.4.1 has the cost of high overheads, both run-time and instrumentation-time. Despite this, CINE manages to instrument the benchmarks in an acceptable time thanks to the refactoring detailed in 4.3.2 and work around run-time overheads with invalidation.

6.1.1 Static

The static framework has very low overheads for simple code. However processing any medium sized binaries incurs an inordinate overhead, which is mainly spent executing trampoline code.

6.1.2 Dynamic

The dynamic alternative adapts well to situations where many calls to short functions would otherwise render CINE unusable. Despite this, it cannot adapt to programs that have intensive thread communication.

6.2 VEX integration

CINE integrates with VEX to bring invalidation policies and function time warps to binary profiling.

6.2.1 Invalidation

The effect of each policy on the run-time overheads varies from program to program. The ”Time” policies have been shown to be less effective than the ”Sample” policies. Invalidation policies must be applied conservatively and with trial and error to avoid worsening CINE’s performance with ”over-invalidation”.

6.2.2 Time warps

Performance engineers are able to accelerate or decelerate an arbitrary number of functions by specifying the function name to CINE. It will register this information with VEX and the following virtual execution will take this into account.

6.3 Future work

6.3.1 Interposing

As mentioned throughout this report, the interposing technique explained in 4.4.2 is worth exploring. It is likely to significantly reduce overheads and adds scope for partial profiling of binaries. This could allow CINE to scale to production systems.

6.3.2 Further VEX integration

VEX provides very useful features, as described in 2.8. I/O simulation, method modeling and multi-core simulation in virtual time binaries would add more detail to CINE's profiles and pose interesting challenges.

6.3.3 Self-instrumenting code

To combine the flexibility of CINE's dynamic approach with the lightness of its static approach, it is worth investigating the possibility of self rewriting code. Several frameworks exist that implement this with the heavyweight JIT approach, [12, 21, 25], however an instrument replacing technique could solve the overhead problems experienced by CINE.

6.3.4 Unified binary VTF framework

CINE makes several assumptions about the libraries, i.e.*pthreads*, and the compiler's, i.e. *gcc*'s, calling conventions that are used when making a multi-threaded binary. An interesting exploit would be to abstract out these dependencies to create a system that could profile any code executing on a known machine. Such a system could work closely with the Linux kernel to provide system-wide profiling or have a plug-in architecture to allow extensions for profiling of any given language.

Appendices

A. Pthread example

```c
/*
* SOURCE: Adapted from example code in "Pthreads Programming", B. Nichols
* et al. O'Reilly and Associates.
*/

#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUM_THREADS 3
#define TCOUNT 10
#define COUNT_LIMIT 12

int count = 0;
pthread_mutex_t count_mutex;
pthread_cond_t count_threshold_cv;

void *inc_count(void *t)
{
  int i;
  long my_id = (long)t;

  for (i=0; i < TCOUNT; i++) {
    pthread_mutex_lock(&count_mutex);
    count++;

    /*
  Check the value of count and signal waiting thread when condition is
  reached. Note that this occurs while mutex is locked.
  */
    if (count == COUNT_LIMIT) {
      pthread_cond_signal(&count_threshold_cv);
      }
    pthread_mutex_unlock(&count_mutex);

    /* Do some work so threads can alternate on mutex lock */
    sleep(1);
    }
  pthread_exit(NULL);
```

```
}

void *watch_count(void *t)
{
  long my_id = (long)t;

  /*
  Lock mutex and wait for signal. Note that the pthread_cond_wait routine
  will automatically and atomically unlock mutex while it waits.
  Also, note that if COUNT_LIMIT is reached before this routine is run by
  the waiting thread, the loop will be skipped to prevent pthread_cond_wait
  from never returning.
  */
  pthread_mutex_lock(&count_mutex);

  while (count < COUNT_LIMIT) {
    pthread_cond_wait(&count_threshold_cv, &count_mutex);
    count += 125;
  }

  pthread_mutex_unlock(&count_mutex);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
  int i, rc;
  long t1=1, t2=2, t3=3;
  pthread_t threads[3];
  pthread_attr_t attr;

  /* Initialize mutex and condition variable objects */
  pthread_mutex_init(&count_mutex, NULL);
  pthread_cond_init (&count_threshold_cv, NULL);

  /* For portability, explicitly create threads in a joinable state */
  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
  pthread_create(&threads[0], &attr, watch_count, (void *)t1);
  pthread_create(&threads[1], &attr, inc_count, (void *)t2);
  pthread_create(&threads[2], &attr, inc_count, (void *)t3);

  /* Wait for all threads to complete */
  for (i = 0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
  }

  /* Clean up and exit */
  pthread_attr_destroy(&attr);
  pthread_mutex_destroy(&count_mutex);
```

```
90    pthread_cond_destroy(&count_threshold_cv);
91    pthread_exit (NULL);
92
93 }
```

Listing A.1: Example synchronizing code

B. Ptrace

```
/* Type of the REQUEST argument to 'ptrace.'  */
enum __ptrace_request
{
  /* Indicate that the process making this request should be traced.
     All signals received by this process can be intercepted by its
     parent, and its parent can use the other 'ptrace' requests.  */
  PTRACE_TRACEME = 0,
#define PT_TRACE_ME PTRACE_TRACEME

  /* Return the word in the process's text space at address ADDR.  */
  PTRACE_PEEKTEXT = 1,
#define PT_READ_I PTRACE_PEEKTEXT

  /* Return the word in the process's data space at address ADDR.  */
  PTRACE_PEEKDATA = 2,
#define PT_READ_D PTRACE_PEEKDATA

  /* Return the word in the process's user area at offset ADDR.  */
  PTRACE_PEEKUSER = 3,
#define PT_READ_U PTRACE_PEEKUSER

  /* Write the word DATA into the process's text space at address ADDR.  */
  PTRACE_POKETEXT = 4,
#define PT_WRITE_I PTRACE_POKETEXT

  /* Write the word DATA into the process's data space at address ADDR.  */
  PTRACE_POKEDATA = 5,
#define PT_WRITE_D PTRACE_POKEDATA

  /* Write the word DATA into the process's user area at offset ADDR.  */
  PTRACE_POKEUSER = 6,
#define PT_WRITE_U PTRACE_POKEUSER

  /* Continue the process.  */
  PTRACE_CONT = 7,
#define PT_CONTINUE PTRACE_CONT

  /* Kill the process.  */
  PTRACE_KILL = 8,
```

```
#define PT_KILL PTRACE_KILL

  /* Single step the process.
     This is not supported on all machines.  */
  PTRACE_SINGLESTEP = 9,
#define PT_STEP PTRACE_SINGLESTEP

  /* Execute process until next taken branch.  */
  PTRACE_SINGLEBLOCK = 12,
#define PT_STEPBLOCK PTRACE_SINGLEBLOCK

  /* Get siginfo for process.  */
  PTRACE_GETSIGINFO = 13,
#define PT_GETSIGINFO PTRACE_GETSIGINFO

  /* Set new siginfo for process.  */
  PTRACE_SETSIGINFO = 14,
#define PT_GETSIGINFO PTRACE_GETSIGINFO

  /* Attach to a process that is already running. */
  PTRACE_ATTACH = 16,
#define PT_ATTACH PTRACE_ATTACH

  /* Detach from a process attached to with PTRACE_ATTACH.  */
  PTRACE_DETACH = 17,
#define PT_DETACH PTRACE_DETACH

  /* Get all registers (pt_all_user_regs) in one shot */
  PTRACE_GETREGS = 18,
#define PT_GETREGS PTRACE_GETREGS

  /* Set all registers (pt_all_user_regs) in one shot */
  PTRACE_SETREGS = 19,
#define PT_SETREGS PTRACE_SETREGS

  /* Continue and stop at the next (return from) syscall.  */
  PTRACE_SYSCALL = 24
#define PT_SYSCALL PTRACE_SYSCALL
};
```

C. Instrumentation of inc_count

```
0000000000400b16 <inc_count>:
  400b16: 55                    push   %rbp
  400b17: 48 89 e5              mov    %rsp,%rbp
  400b1a: 48 83 ec 20           sub    $0x20,%rsp
  400b1e: 48 89 7d e8           mov    %rdi,-0x18(%rbp)
  400b22: 48 8b 45 e8           mov    -0x18(%rbp),%rax
  400b26: 48 89 45 f0           mov    %rax,-0x10(%rbp)
  400b2a: c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)
  400b31: e9 8d 00 00 00        jmpq   400bc3 <inc_count+0xad>
  400b36: bf 20 15 60 00        mov    $0x601520,%edi
  400b3b: e8 d0 fe ff ff        callq  400a10 <pthread_mutex_lock@plt>
  400b40: 8b 05 be 09 20 00     mov    0x2009be(%rip),%eax        # 601504 <
    count>
  400b46: 83 c0 01              add    $0x1,%eax
  400b49: 89 05 b5 09 20 00     mov    %eax,0x2009b5(%rip)        # 601504 <
    count>
  400b4f: 8b 05 af 09 20 00     mov    0x2009af(%rip),%eax        # 601504 <
    count>
  400b55: 83 f8 0c              cmp    $0xc,%eax
  400b58: 75 30                 jne    400b8a <inc_count+0x74>
  400b5a: 8b 15 a4 09 20 00     mov    0x2009a4(%rip),%edx        # 601504 <
    count>
  400b60: 48 8b 45 f0           mov    -0x10(%rbp),%rax
  400b64: 48 89 c6              mov    %rax,%rsi
  400b67: bf 98 0e 40 00        mov    $0x400e98,%edi
  400b6c: b8 00 00 00 00        mov    $0x0,%eax
  400b71: e8 aa fd ff ff        callq  400920 <printf@plt>
  400b76: bf 60 15 60 00        mov    $0x601560,%edi
  400b7b: e8 e0 fd ff ff        callq  400960 <pthread_cond_signal@plt>
  400b80: bf d2 0e 40 00        mov    $0x400ed2,%edi
  400b85: e8 76 fd ff ff        callq  400900 <puts@plt>
  400b8a: 8b 15 74 09 20 00     mov    0x200974(%rip),%edx        # 601504 <
    count>
  400b90: 48 8b 45 f0           mov    -0x10(%rbp),%rax
  400b94: 48 89 c6              mov    %rax,%rsi
  400b97: bf e8 0e 40 00        mov    $0x400ee8,%edi
  400b9c: b8 00 00 00 00        mov    $0x0,%eax
  400ba1: e8 7a fd ff ff        callq  400920 <printf@plt>
  400ba6: bf 20 15 60 00        mov    $0x601520,%edi
```

```
  400bab: e8 f0 fd ff ff        callq  4009a0 <pthread_mutex_unlock@plt>
  400bb0: bf 01 00 00 00        mov    $0x1,%edi
  400bb5: b8 00 00 00 00        mov    $0x0,%eax
  400bba: e8 31 fe ff ff        callq  4009f0 <sleep@plt>
  400bbf: 83 45 fc 01           addl   $0x1,-0x4(%rbp)
  400bc3: 83 7d fc 09           cmpl   $0x9,-0x4(%rbp)
  400bc7: 0f 8e 69 ff ff ff     jle    400b36 <inc_count+0x20>
  400bcd: bf 00 00 00 00        mov    $0x0,%edi
  400bd2: e8 d9 fd ff ff        callq  4009b0 <pthread_exit@plt>


0000000000400b16 <inc_count>:
  400b16: e9 4c 0b 30 00        jmpq   701667 <inc_count_dyninst>
  400b1b: 83 ec 20              sub    $0x20,%esp
  400b1e: 48 89 7d e8           mov    %rdi,-0x18(%rbp)
  400b22: 48 8b 45 e8           mov    -0x18(%rbp),%rax
  400b26: 48 89 45 f0           mov    %rax,-0x10(%rbp)
  400b2a: c7 45 fc 00 00 00 00  movl   $0x0,-0x4(%rbp)
  400b31: e9 8d 00 00 00        jmpq   400bc3 <inc_count+0xad>
  400b36: e9 af 0b 30 00        jmpq   7016ea <inc_count_dyninst+0x83>
  400b3b: e8 d0 fe ff ff        callq  400a10 <targ400a10>
  400b40: e9 af 0b 30 00        jmpq   7016f4 <inc_count_dyninst+0x8d>
  400b45: 00 83 c0 01 89 05     add    %al,0x58901c0(%rbx)
  400b4b: b5 09                 mov    $0x9,%ch
  400b4d: 20 00                 and    %al,(%rax)
  400b4f: 8b 05 af 09 20 00     mov    0x2009af(%rip),%eax        # 601504 <
    count>
  400b55: 83 f8 0c              cmp    $0xc,%eax
  400b58: 75 30                 jne    400b8a <inc_count+0x74>
  400b5a: e9 b3 0b 30 00        jmpq   701712 <inc_count_dyninst+0xab>
  400b5f: 00 48 8b              add    %cl,-0x75(%rax)
  400b62: 45                    rex.RB
  400b63: f0 48 89 c6           lock mov %rax,%rsi
  400b67: bf 98 0e 40 00        mov    $0x400e98,%edi
  400b6c: b8 00 00 00 00        mov    $0x0,%eax
  400b71: e8 aa fd ff ff        callq  400920 <targ400920>
  400b76: e9 b3 0b 30 00        jmpq   70172e <inc_count_dyninst+0xc7>
  400b7b: e8 e0 fd ff ff        callq  400960 <targ400960>
  400b80: e9 b3 0b 30 00        jmpq   701738 <inc_count_dyninst+0xd1>
  400b85: e8 76 fd ff ff        callq  400900 <targ400900>
  400b8a: e9 b3 0b 30 00        jmpq   701742 <inc_count_dyninst+0xdb>
  400b8f: 00 48 8b              add    %cl,-0x75(%rax)
  400b92: 45                    rex.RB
  400b93: f0 48 89 c6           lock mov %rax,%rsi
  400b97: bf e8 0e 40 00        mov    $0x400ee8,%edi
  400b9c: b8 00 00 00 00        mov    $0x0,%eax
  400ba1: e8 7a fd ff ff        callq  400920 <targ400920>
  400ba6: e9 b3 0b 30 00        jmpq   70175e <inc_count_dyninst+0xf7>
  400bab: e8 f0 fd ff ff        callq  4009a0 <targ4009a0>
  400bb0: e9 b3 0b 30 00        jmpq   701768 <inc_count_dyninst+0x101>
```

```
400bb5: b8 00 00 00 00        mov    $0x0,%eax
400bba: e8 31 fe ff ff        callq  4009f0 <targ4009f0>
400bbf: cc                    int3
400bc0: 45 fc                 rex.RB cld
400bc2: 01 e9                 add    %ebp,%ecx
400bc4: b4 0b                 mov    $0xb,%ah
400bc6: 30 00                 xor    %al,(%rax)
400bc8: 8e 69 ff              mov    -0x1(%rcx),%gs
400bcb: ff                    (bad)
400bcc: ff e9                 ljmpq  *<internal disassembler error>
400bce: b4 0b                 mov    $0xb,%ah
400bd0: 30 00                 xor    %al,(%rax)
400bd2: e8 d9 fd ff ff        callq  4009b0 <targ4009b0>
```

D. Prediction testing program

```
#include <ctime>
#include <iostream>
#include <math.h>
#include <string>
#include <unistd.h>
#include "pthread.h"

using namespace std;

  long load;
  double temp = 1.0;
  bool testTimeOut = false;

  void *RunningThread(void *t){
    for (int j = 0; j < 200; j++){
      for (int i = 1; i<load; i++) {
        temp += pow(1.0 + 1.0/i, i);
      }
      temp = 0;
    }
  }

  int main(int argc, char *argv[]) {
    int threads = stoi(argv[1]);
    load = stol(argv[2]);
    pthread_attr_t attr;
    pthread_attr_init(&attr);

    pthread_t thread[threads];
    pthread_t timer;

    int round = 0;
    time_t start;
    time(&start);

      long i;
      for (i = 0; i < threads; i++) {
        pthread_create(&thread[i], &attr, RunningThread, (void *)i);
      }
```

```
40        for (i = 0; i < threads; i++) {
41          pthread_join(thread[i], NULL);
42        }
43        cout << "Round " << (++round) << " finished after " << (time(NULL)-start)
        << endl;
44    }
```

Listing D.1: Test code to generate some computational load. the first argument is a number of threads to spawn and the second argument is an arbitrary ”load” number

Bibliography

[1]

[2] Dyninst programmers guide, March 2013.

[3] Andrew. http://floodyberry.wordpress.com/2009/10/07/high-performance-cplusplus-profiling/, 2014.

[4] Nikolaos Baltas. Software performance engineering using time program execution. Phd thesis, Imperial College London.

[5] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] GNU Binutils. http://www.gnu.org/software/binutils/, 2014.

[7] Boost. http://www.boost.org/, 2014.

[8] Clang/LLVM. http://clang.llvm.org/, 2014.

[9] Google Concurrency. http://code.google.com/p/google-concurrency-library/, 2014.

[10] Timothy W Curry et al. Profiling and tracing dynamic library usage via interposition. In *USENIX Summer*, pages 267–278, 1994.

[11] Delorie. http://www.delorie.com/gnu/docs/gdb/gdbint_toc.html, 2014.

[12] Qin Zhao Derek Bruening. Building dynamic instrumentation tools with dynamorio. 2011.

[13] Die.net. http://linux.die.net/man/3/dlopen, 2014.

[14] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux. 2003.

[15] GNU gcc. https://gcc.gnu.org/, 2014.

[16] GNU gdb. www.gnu.org/s/gdb/, 2014.

[17] GNU gprof. http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html, 2014.

[18] David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, 1985.

[19] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183. IEEE, 2010.

[20] John Levon and Philippe Elie. Oprofile: A system profiler for linux, 2004.

[21] 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. In *Acm Sigplan Notices*, volume 40, pages 190–200. ACM, 2005.

[22] Dyninst mail archives. https://www-auth.cs.wisc.edu/lists/dyninst-api/2013/msg00077.shtml, 2014.

[23] Open MPI. http://www.open-mpi.org/, 2014.

[24] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74. ACM, 2007.

[25] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[26] OpenMP. http://openmp.org/wp/, 2014.

[27] Shiny. http://sourceforge.net/p/shinyprofiler/, 2014.

[28] Sourceforge Elf startup. http://asm.sourceforge.net/articles/startup.html, 2014.

[29] Unknown. http://blog.csdn.net/u010838822/article/details/14094287, 2014.