



MENG INDIVIDUAL PROJECT

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Leveraging Linux kernel tracing to classify and detail application bottlenecks

Author:

Ashley J Davies-Lyons

Supervisor:

Dr. Anthony Field

Second Marker:

Dr. Giuliano Casale

June 17, 2019

Abstract

GAPP is a bottleneck identification tool that uses Linux kernel probes to identify periods of reduced parallelism in multithreaded programs. Although GAPP is effective at identifying the lines of source code that lead to a bottleneck, it is unable to classify the type of bottleneck - for example whether it is due to lock contention or I/O.

This project solves this problem by augmenting the stack traces generated by GAPP with classifications, and adds details of any files that were interacted with, or IP addresses that were interacted with. Additionally, by tracking kernel-level synchronisation (‘futex’) calls, we develop a lock analysis feature that assists with identifying particularly critical locks (and unlockers) in user applications. Further, we provide a summary of the most critical *individual* file and synchronisation actions. In the spirit of GAPP, we implement this without requiring instrumentation, and does not introduce any language or library dependencies.

We find that our extended tool is able to reliably classify the different categories of bottleneck, and adds useful information to the GAPP output which is useful in diagnosing the root causes of a bottleneck. We verify this with two large open source projects - an image tracking benchmark, and a production game server. Finally, we find that the overhead we add is competitive with similar tools, and that our tool works correctly with alternative threading library - having evaluated with TBB and `pthread`s.

In addition to our main contributions, we additionally add a number of quality-of-life improvements to the tool, including a user interface to present the data, improved stack trace reporting, and easier methods of attaching to user applications.

Acknowledgements

I am hugely thankful to my supervisor, Tony, for being consistently helpful, supportive, and available for meetings over the last year, in addition to being a great personal tutor for the past four years, and also to my co-supervisor, Reena, for being ever-willing to help with strange bugs on short notice, and whose extensive body of relevant knowledge and experience saved me an incalculable amount of debugging and stress.

I'd also like to acknowledge my tremendous gratitude to my sixth form lecturers Gareth and Jonathan, whose eagle-eyed spotting of a mistake in my A-Level results rescued my place to study here in the first place.

And, last but not least, I am grateful to my mother, the rest of my family, & my friends, who have been endlessly supportive through the last few months.

Contents

1	Introduction	6
1.1	Objectives	7
1.2	Contributions	8
2	Background	9
2.1	Note on threading	9
2.2	Software Performance	9
2.2.1	Tracing	10
2.2.2	Sampling	10
2.2.3	Categorising performance analysis	11
2.3	Task-based parallelism - TBB	12
2.4	Overview of recent profiling tools and approaches	12
2.4.1	wPerf	12
2.4.2	Coz, and causal profiling	13
2.4.3	TaskProf	13
2.4.4	GAPP	13
2.4.5	Comparison of GAPP and wPerf	14
2.5	Synchronisation	14
2.5.1	Background	14
2.5.2	Processor-level synchronisation	14
2.5.2.1	Summary for x86	15
2.5.3	Synchronisation primitives	15
2.5.3.1	Types of primitives	15
2.5.3.2	Spinning	16

2.5.4	Futexes	17
2.6	eBPF: tracing in the Linux kernel	19
2.6.1	BCC	20
2.6.2	Performance of eBPF	21
3	Extending GAPP	23
3.1	Overview	23
3.1.1	Goal	23
3.1.2	Implementation	23
3.1.3	Feature summary	24
3.1.4	Comparison	25
3.1.5	Backend summary	26
3.2	Causation flag system	27
3.3	Identifying Synchronisation Bottlenecks	29
3.3.1	Limitations of return probes	29
3.3.1.1	Avoiding this issue	31
3.3.2	Summary	32
3.3.3	Tracking futex wait operations	33
3.3.4	Tracking futex wake operations	34
3.4	Identifying IO Bottlenecks	35
3.4.1	Identifying IO Bottlenecks - Files	36
3.4.2	Identifying IO Bottlenecks - Networking	38
3.4.3	Identifying IO Bottlenecks - Reads and Writes	38
3.5	Modifications to the core GAPP algorithm	39
3.6	Python front-end processing	40
3.6.1	Additional synchronisation stack traces	40
3.6.2	Lock page	41
3.6.3	Most Critical Futex Activity	41
3.6.4	Most Critical File Activity	42
3.7	User Experience	43
3.7.1	Tracing relevant threads	43
3.7.2	Enhanced Stack Trace Reporting	44

3.7.3	User Interface	45
3.8	Engineering larger systems with BCC and eBPF	46
3.8.1	Separation of code	46
3.8.2	Debugging issues	49
4	Evaluation	51
4.1	Setup	51
4.2	Note on glibc stack traces and parent threads	51
4.3	Individual feature evaluation	52
4.3.1	Elementary synchronisation	52
4.3.2	Shared locks	55
4.3.3	Multiple locks	57
4.3.4	IO - File reading	59
4.3.5	IO - File writing	62
4.4	Alternative threading library	64
4.4.1	Ad-hoc threading	64
4.4.2	TBB	65
4.5	Real-world Benchmarks & Programs	68
4.5.1	Parsec - Bodytrack	68
4.5.2	Cuberite - A Minecraft Server	71
4.6	Quantifying errors	75
4.6.1	Evaluating error count in Cuberite	76
4.6.2	Missing futex traces	76
5	Conclusion	77
5.1	Summary	77
5.2	Future Work	77
5.2.1	DNS Probing	77
5.2.2	Conflation of identically-defined locks	78
5.2.3	Ad-hoc synchronisation	78
5.2.4	Unit testing	78
A	Additional Screenshots	82

1 | Introduction

Improvements in the processing power of a single core have been considerably slowing in the past decade, compared to the prior decades. One of the most powerful consumer desktop CPUs today, the Intel i9-9900KF [1], ships with a 3.6GHz clock speed as standard; little more than the 3.46GHz of the Intel i7-990X [2] from 2011. Instead, core counts are increasing, and manufacturers are implementing technologies such as SMT (Simultaneous Multithreading, a.k.a. *hyperthreading*), which enable an individual core to run multiple threads concurrently. Almost every modern CPU [3] offers at least four to eight cores, and even a single chip made for a high performance server can contain dozens of cores. Accordingly, software applications are transitioning towards models which take advantage of multiple processor cores.

From a developer's perspective, the runtime operations of multi-threaded applications are generally harder to reason about than an equivalent single-threaded application. Additionally, traditional performance metrics and measurement tools designed for single-threaded applications are not as effective when applied to multi-threaded programs, because of extra inefficiencies and forms of bottlenecks which are introduced by the parallelism. Most obviously, there is the direct overhead introduced by the threading itself - for example, context switches caused by moving threads on and off CPU cores can add significant overhead to an otherwise efficient application. This overhead is important, but by far the most significant performance issues arise due to suboptimal thread synchronisation.

When threads wait on each-other (or even on IO devices such as a disk) when they could be doing other work, a program can suffer significant performance losses. On an eight core machine, an application with eight threads can perform worse than a single threaded version if there is significant contention for a single resource, or if locking is too coarse. Fixing these sorts of issues can be tricky, but it is often just as tricky to find the cause in the first place: it is far from trivial to know in advance which parts of an application will cause performance issues, and this difficulty is only worsened when we start trying to reason about the runtime of multithreaded software with complex inter-thread interactions.

Considering all of this, it is clear that there is a need for effective, straightforward, and accurate tools for developers to profile and diagnose inefficiencies in their applications. They must be effective so that they can help achieve large performance improvements, straightforward in order to enable mass-adoption by developers, and accurate so as to avoid developers losing faith in the tool.

1.1 Objectives

There exists a wide body of research focused on identifying bottlenecks in multithreaded applications [4, 5, 6, 7], much of which assumes the use of a specific threading library [6] or focuses on a specific concurrency model such as task-parallel programming (Intel’s TBB, Cilk, etc.) [5]. Research which takes a more general approach is fairly recent [4, 7, 8].

This project extends the bottleneck detection tool GAPP [8], which detects synchronisation bottlenecks using a generic approach that avoids introducing a dependency on a specific threading library or language, and does not require the program under analysis to be instrumented in any way.

In its present form, GAPP utilises tracing features in the kernel (its inner workings are described further in [subsection 2.4.4](#)) to identify when a significant number of threads in an application are in a non-runnable state (e.g. waiting on a lock), and while this is true, any application threads that are descheduled have a stack trace sample taken and reported. Additionally, there is periodic sampling of the threads to accumulate a set of critical functions and lines which were executing by the thread prior to being descheduled. The reported stack traces are weighted by the ratio of non-runnable threads, as well as the time this state has been maintained for, to obtain their *criticality metric*.

The primary objective of this project is to take this idea further, and use additional tracing to classify detected bottlenecks into broad categories, such as synchronisation and forms of IO, as well as to deduce additional information that increases the utility of the reported stack traces, such as related file names of an IO operation, where it is viable to do so.

This is important as it is often desirable to look into specific forms of bottlenecks in an application. For example, if a lock is highly contended, it is often possible to make the lock more granular to achieve higher parallelism. On the other hand, if threads are bottlenecked by a single IO device, it might not be so straightforward to solve, and improved hardware might be required to improve throughput or otherwise offload some of the work, which can be expensive and require more investment. In this example, a developer may be willing to accept the slow IO, but may still wish to identify other concurrency bottlenecks – or vice versa – and we aim to enable this.

In summary, with these goals in mind we aim to produce a tool which can, given a Linux binary, and without applying any modifications to the source code or having advance knowledge of what threading libraries it utilises, succeed in identifying, categorising and detailing concurrency bottlenecks in a meaningful way.

1.2 Contributions

Our contributions build directly on top of GAPP:

- A classification system which is able to reliably and correctly, based on our evaluation, classify the output critical stack traces ([section 3.3](#), [section 3.4](#) [section 3.5](#)).
- Additional kernel- and glibc- level tracing to add additional utility to stack traces, such as which files they were interacting with, or which stack traces woke them from synchronisation sleep (sections as above), as well as a summary of all locks weighted by importance ([subsection 3.6.1](#), [subsection 3.6.2](#), [subsection 3.6.3](#), [subsection 3.6.4](#)).
- User experience improvements in the tool - adding a user interface, cleaner and more helpful stack traces, and easier ways to trace programs ([section 3.7](#)).
- Evaluation against a number of contrived programs, which demonstrates the ability of our extended tool to classify traces, work on different threading approaches, and provide genuinely useful information to help resolve synchronisation issues ([section 4.3](#)). We demonstrate that our tool works on different threading libraries, including `pthread`s, TBB, and ad-hoc threading using system calls.
- Evaluation against a number of real-world programs, including a component of the Parsec [9] benchmark and server software for the popular game Minecraft [10] ([section 4.5](#)). We show a competitive overhead of around 5%, comparable to similar tools, and that our tool is capable of handling real-world applications correctly and providing valuable utility for debugging parallelism issues in such programs.

2 | Background

2.1 Note on threading

Threads can be implemented and managed both in kernel-space and in user-space. There are advantages and disadvantages to both approaches, with user-space threads offering some useful benefits in higher-level languages like Ruby and Python¹, but kernel-space threading often making better usage of multi-core CPUs. Kernel-space threading is typically used more in lower-level compiled languages like C and C++.

For the purposes of this project, we are only interested in kernel-space threading, and where we have not directly referred to threads as user-space threads, the term *thread* refers to kernel-space threads.

2.2 Software Performance

For many projects, the importance of software performance is neglected until late in the development cycle [11]. This is despite the fact that users expect newer software to do more, be more responsive, and finish work faster. Meeting these goals can be achieved in a number of ways, with the most obvious being to carefully engineer the system from the start to achieve high performance.

There are some issues with such an approach - notably that it is harder to change the software mid-development if you design it all up front, which conflicts with another increasingly common user expectation: the ability to adapt software to feedback in an agile fashion. An even bigger issue, though, is that it's very difficult to identify performance bottlenecks in advance; especially when you consider multithreaded, modern systems, with many layers of abstractions and complexities. As a result, performance issues are typically discovered when the application is operational, but not performing as well as users wish or the business requires. In fact, most software doesn't perform as well as it could - often by significant amounts. It is not uncommon to see research which contributes a novel way to identify bottlenecks in a program, and for the authors to then identify and sometimes even fix deficiencies in mainstream applications which cause significant speedups. For example:

1. In [7], the authors managed to speed up `memcached` by 9.4% and `SQLite` by 25%

¹For example, it is often desirable to have synchronous I/O. The interpreter can create the illusion of synchronous I/O operations while using asynchronous I/O under the hood, and instead run another thread while a thread has asynchronous I/O running

2. In [6], the authors identified an issue in the quantum chemistry application **MADNESS**. They show it is a primary cause of notably diminishing returns on the number of cores, where increasing from 4 to 16 cores only gives a speedup of 2.2x. The maintainers knew about the diminishing returns, but not the cause.
3. In [12], the authors identified three previously undiscovered performance bugs in **MySQL** and **memcached**.

There are more examples, but the above is suffice to show that there is an opportunity for tools to discover previously unknown bottlenecks in major software such as **MySQL**. Since this is the case, it's reasonable to conclude that software which is used less, and hasn't had the same level of attention from performance engineers and researchers, likely has even larger opportunities for improvement.

2.2.1 Tracing

Tracing in software engineering has some notable similarities to logging: at specific points in the execution of a program, tracing information is emitted, and then it is later analysed by engineers. A core difference between the two concepts is that tracing data is largely consumed by developers' tooling for the purposes of debugging or profiling, whereas logging is often targeted towards maintainers and designed for human consumption.

Tracing can be a good way for tools to gather data about a program's execution for profiling its performance but, unfortunately, the price to pay to acquire this data commonly involves manually inserting tracing points in an application, which is not always viable or desirable. For some things, such as domain-specific progress tracing - for example, tracing the start/end of processing a query in a SQL database, so that data like latency can be calculated - it is going to be inevitable: a generic tool will not know what 'progress' means for a given application without some manual assistance. On the other hand, it is reasonable to expect that tracing for points which can be well-defined in a general-purpose tool, such as 'all disk accesses', should be achievable without manual effort for each application.

Linux offers the ability to trace specific functions, including almost any kernel-level function, without having to instrument them manually, which is exposed through eBPF (explored in [section 2.6](#)).

2.2.2 Sampling

In contrast to tracing, another method of deriving information about a program's execution is through *sampling*. Sampling involves polling information about execution periodically, and later using the information to derive statistics about a program's execution.

For example, sampling the instruction pointer of a running application every few milliseconds can allow us to determine execution bottlenecks of a system - if running software is regularly found to be running within a matrix multiplication method, then it is reasonable to assume that, at least for a single-threaded application, optimising that method would result in the software as a whole performing better.

In a blog post [13], Dan Luu comments that a deficiency in the approach that sampling profilers take arises due to how they aggregate results as averages. This is not always good enough for some use cases, and he gives a specific example at Google. They are often interested in tail latencies, which are the ones more likely to lose or frustrate customers and are often very significant in their performance difference - in a graph of remote procedure calls, the slowest two (of nearly a hundred) calls were more than twice as slow as the rest. Since tracing reliably captures all events of interest, it can provide more nuanced analysis for areas of interest, which can be of benefit in certain situations such as Google's need to analyse tail latency.

2.2.3 Categorising performance analysis

Tools which analyse performance of multithreaded applications can be identified as either conducting on-CPU analysis or off-CPU analysis [14]. Those which conduct on-CPU analysis are closer to traditional single-threaded analysis, which find *execution* bottlenecks - i.e. they point us towards those parts of the code which are taking the most on-CPU time. In contrast, off-CPU analysis is exclusively used for multithreaded applications, and aims to identify those bottlenecks which arise from *waiting*. Such bottlenecks may arise from, for example, a high contention lock, or slow synchronous I/O. Software may have both types of bottlenecks, so it not always going to be possible to find all performance issues with a single tool.

Another important class of profilers is those which utilise an approach known as *call path profiling* [15]. The central idea behind this is that sometimes context is required to understand why (or even if) parts of a program are bottlenecks. For example, imagine a subroutine for acquiring a lock. A tool which told us that a substantial amount of waiting time is spent parked in this subroutine is not particularly useful or insightful - of more importance is *which* code is calling into this subroutine? With this additional context, we can learn which locks are causing the most waiting, and then put effort into, for example, increasing the granularity of that lock. So, by considering calling context, tools can often be more specific about which parts of the code are causing bottlenecks, and hence produce more useful analysis [16]. It is not always true that calling context is relevant to a bottleneck, since some parts of the codebase may indeed just be detrimental to performance, regardless of further context, but these parts of the application should still be identified since they apply to all contexts.

2.3 Task-based parallelism - TBB

Many engineers choose to build their software on top of libraries, which abstract away some of the difficulties and minimise the thread-related boilerplate of writing parallel code by creating more powerful abstractions on top of threads.

For example, task-based parallel programming is an alternative to thread-based parallelism. In this model of programming, programmers encode details about *tasks* that need to be done and information about data dependencies, and a runtime scheduler will do the work of assigning these tasks to threads and balancing the work across the processor. This is often much easier than having to manually write threading code, and has the additional benefit that the scheduling engine can switch tasks on- and off- threads when the tasks are blocked.

In particular, Intel’s *TBB* (Threaded Building Blocks) library is of interest for this project. TBB allows programmers to define loops which can be parallelised, specify how iterations should be ‘chunked’ together to avoid data races, define data pipelines and more. While there can be significant overhead from the scheduler with a lot of cores (notable performance degradation has been shown for, for example, 32 cores [17]), it remains a helpful and widely used library.

2.4 Overview of recent profiling tools and approaches

2.4.1 wPerf

wPerf [4] conducts off-CPU analysis to find “critical waiting events that limit the maximal throughput of multi-threaded applications.” Importantly, it takes a general approach, and does not require the use of specific threading library such as `pthread`s.

A key insight is that it distinguishes between local impact and global impact of waiting events – the local impact is the time lost by threads actively waiting on that thread, and the global impact is the knock-on effect this has on all of the other threads. The authors model I/O devices as threads, which allows a general approach that can identify both bottlenecks caused by I/O devices and also inter-thread waiting.

A difficulty that arises from looking at waiting events is that sometimes the cause of the waiting is unclear. For example, consider three threads, with one active and two sleeping. If the active thread wakes one of the sleeping threads, and later the newly awoken thread wakes the third, then it can be unclear during analysis whether the thread which was initially active created a scenario where either sleeping threads could be woken up (e.g. a random lock), and it just happened to be that the first thread to wake was chosen by the scheduler (this is referred to as *symmetric waiting*), or if the final thread to wake was

woken directly by the second thread (*asymmetric waiting*). The authors of [4] decided that, since it may be responsible for the additional waiting of the third thread, the thread that wakes first must have its waiting time weighted in the analysis to account for this possibility, and the final thread to wake was considered to be waiting exclusively on the first to wake for the whole time.

2.4.2 Coz, and causal profiling

Coz [7] is a particularly interesting profiling tool as it is able to make precise judgements about how much a bottleneck is slowing down an application. It does this through *causal profiling*, which it uses to view the potential effects on both throughput and latency (though the throughput analysis does require manually inserted progress points in the application).

Coz uses *virtual speedups* to determine this information, which involves pausing all threads but one to create the appearance of the one thread working faster than it is. A notable advantage of an approach like this is that it is able to account for when a bottleneck is not a good choice of optimisation. An example given in [7] is for loading screens: a loading screen can naively seem like a bottleneck, and may be reported by some profilers, since it takes at least as long as the action it is waiting on. However, they are not part of the critical path, and optimising them will rarely be of any use in improving the performance of the application.

2.4.3 TaskProf

TaskProf is a profiler focused on task parallel programs [5], and works exclusively for Intel’s TBB. It measures asymptotic parallelism, and takes a causal approach like Coz’s, which “allows users to estimate improvements in parallelism when regions of code are optimized even before concrete optimizations for them are known.”

2.4.4 GAPP

GAPP [8], the tool that this project is built upon, is based on the premise that serialisation bottlenecks can be detected by paying attention to the number of runnable threads. When the fraction of runnable threads dips below a certain threshold, it is considered critical and a stack trace is reported, weighted by the length of time the offending thread has been running, as well as inversely by the number of threads running - a situation with few threads running for a long time is very critical, whereas a situation with many threads running for a short time is not at all critical.

This numeric quantification of criticality is referred to as the *criticality metric*, and used widely by this project to weight things associated with stack traces such as which

filenames are associated with stack traces.

GAPP uses kernel-level tracing to achieve this, by probing the `sched_switch` trace-point.

In addition, GAPP reports a number of critical functions and lines associated with each stack trace. These are determined by periodic sampling, and linked to the next critical stack trace reported for that thread.

2.4.5 Comparison of GAPP and wPerf

GAPP has similarities to wPerf, which also does not introduce dependencies on libraries like `pthread`s, but works in a slightly different way - GAPP tracks scheduling at a different level, and considers any scheduling event to be a potentially critical one - all of which is deterministic at the time of an individual switch - whereas wPerf works by specifically tracking interrupts and a lower-level switching operation, builds wake-up graphs, then uses post-processing to try to identify possible problems. As a result, the post-processing overhead of GAPP is much lower.

In addition, GAPP reports stack traces directly from the kernel-level tracing mechanisms, which wPerf does not do - it instead associates stack traces which are sampled periodically with Perf. This means GAPP's stack traces can be more reliable and correspond more directly to the pieces of user code causing bottlenecks.

2.5 Synchronisation

2.5.1 Background

Correct multithreaded code that does useful work usually relies on having a method of synchronisation between threads. For example, aggregating data from multiple threads or outputting to a shared device (e.g. a single terminal) both require synchronisation to avoid races - even if it is as simple as waiting for a thread to terminate.

2.5.2 Processor-level synchronisation

In order to implement synchronisation primitives, it is common to utilise atomic instructions. These instructions ensure that relevant memory accesses and writes do not suffer from interference from parallel processing. For example, two concurrent threads which increment a value could result in a single increment if they both read the value before either computes the increment and writes it - they have a data race. There exist

formal models for reasoning about data races, but they are outside of the scope of this project.

A non-exhaustive set of atomic techniques follows:

1. CAS (*Compare And Swap*) is a common approach to avoid the problems described above. A compare and swap operation specifies the current value, tests the value has not changed, then writes the new value - all while ensuring parallel processors are not writing or reading to the value.

On x86 processors, this can be achieved with a `LOCK CMPXCHG` instruction.

2. FAA (*Fetch And Add*) involves atomically adding to a piece of memory, and is achieved on x86 through the use of `LOCK ADD`.
3. TAS (*Test And Set*) is a less general version of CAS, and is typically restricted to *setting* (i.e. to 1). It can be achieved on x86 using `LOCK BTS`

2.5.2.1 Summary for x86

It is worth noting that on x86 systems, all of the instructions mentioned above require the `LOCK` modifier. This is necessitated as it ensures that the relevant memory addresses are not observed or written to by parallel processors in the meantime. Without it, these instructions are only atomic to an individual core - it acts atomically on a single processor system provided there are no external observers of the memory - but together with the modifier, the instructions act atomically even in an SMP processor.

An interesting note about the `LOCK` modifier is that it only works in combination with a small set of instructions. These are listed exhaustively: `BTS`, `BTR`, `BTC`, `XADD`, `CMPXCHG`, `CMPXCHG8B`, `XCHG`, `INC`, `DEC`, `NOT`, `NEG`, `ADD`, `ADC`, `SUB`, `SBB`, `AND`, `OR`, `XOR` [18, p. 2941]. The `XCHG` instruction is implicitly `LOCK`ed; the others, by default, are not.

2.5.3 Synchronisation primitives

In software, a number of abstractions have been built over atomic instructions to make synchronisation more approachable, and intuitive. The most prominent are locks, semaphores, and barriers.

2.5.3.1 Types of primitives

A summary of some of the most common synchronisation primitives follows:

```

void waitForWork() {
    while (!workDone);
    return;
}

void doWork() {
    ...
    workDone = true;
}

```

Listing 2.1: A minimal example of synchronising threads through spinning.

1. **Locks:** a synchronisation primitive which enable mutual exclusion in an area of code, such that no two threads may be executing in certain regions of code at the same time. To enter a *critical section* (the area of code which requires mutual exclusion), the lock must be acquired. It is then released upon leaving, enabling another thread to acquire the lock and enter the section of code.
2. **Semaphores:** in essence, a generalised lock. A semaphore has two operations: increasing and decreasing a counter. Typical semaphore semantics force threads which wish to decrease the counter to wait when the variable is zero, but decrement it and continue when it is non-zero. They then increase the value when they are finished, allowing another thread to begin work. They are most commonly used for guarding resources which a limited number of threads can be working with at a certain time.

A lock is effectively a semaphore whose value is limited to zero and one - zero implying that the lock has been acquired, one that it is free to take.

3. **Condition variables:** the idea of a condition variable is to give a variable an abstract semantic meaning, such as "the list is empty". Threads can then wait on a condition variable, and other threads can wake them when they are met.

2.5.3.2 Spinning

Synchronisation methods can be further classified based on whether or not they *spin* (busy wait). Synchronisation methods which spin are called such because they repeatedly check a condition in some form of loop, as in [Listing 2.1](#), where a thread can call `waitForWork` to wait until another thread has executed `doWork`. In most situations, this approach to synchronisation is not ideal - it uses a full logical CPU core, but achieves no useful work until `workDone` is true.

For anything that might take more than a few cycles to synchronise, it is better to avoid this spinning, and allow the CPU to do useful work on other threads instead. This

becomes even more important when you consider that doing repeated tests on a variable in this manner consumes bandwidth between physical cores [19], which can cause it to slow down other multiprocessing going on in the system *on top of* consuming a whole core.

A straightforward way to waste less CPU time is to keep the loop, but instead of spinning indefinitely, yield the processor for some period of time. This approach also has some issues, however - since the thread can underestimate the time, it can be re-scheduled before the condition is met, and get switched on-and-off a CPU core, damaging performance through repeated context switches. Alternatively, it can do the opposite: over-estimating the time the task will take, sleeping overzealously, and wasting time on what could potentially be a critical path of the application.

To improve on this situation, user programs on modern operating systems are able to notify the kernel (and, by extension, the scheduler) that a thread is waiting on some condition. This allows the thread to willingly yield the CPU until another thread indicates the condition may be met. Usually, this is a low-level operation which is abstracted from the majority of developers by a library such as pthreads.

2.5.4 Futexes

In 2.5.3.2, it was concluded that it would be helpful to be able to conditionally yield threads in order to avoid spinning for synchronisation. This can be achieved in Linux through the use of the `futex` system call. This system call accepts the address of a user-space integer value, which is referred to as the futex (*fast user-space mutex*). The system call can perform a wide variety of actions based on them, depending on the parameters passed into it. The operations the system call performs have been described in detail by Ulrich Drepper [20], but an abridged description of the key ones follows:

1. `FUTEX_WAIT`: Given a futex (address) and a value, blocks the thread if the value stored in the futex is equal to the value provided.
2. `FUTEX_WAKE`: Wakes a specified number (usually one or all) of the threads waiting on a specific futex.

There are additional parameters to the system call, which allow functionality such as a timeout to be applied to a wait [20], but describing these in depth is outside the scope of this report.

Futexes are more general than a simple lock, since any value can be waited on - for example, they can be used to implement semaphores. Additionally, while we talk about a thread ‘holding’ a futex, in reality there is no such requirement - a thread needn’t set a futex to a certain lock value to later change it back. However, the notion of a simple

lock is a very common usage pattern and these strange edge cases can still, in some way, be viewed through the lens of a more abstract ‘lock’ mechanism - ultimately some threads will `futex_wait` on a futex and some will `futex_wake` those that are waiting.

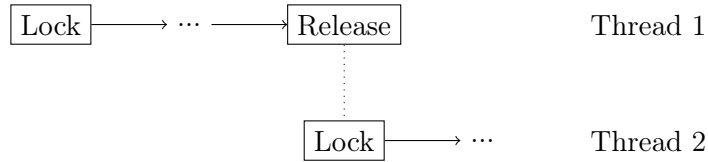


Figure 2.1: Example of a lock without contention; no futex system calls are made.

An important nuance to note is that many futex operations can be performed in user-space: user code can use a method such as CAS primitives to determine if there is contention (e.g. if a thread is waiting on a lock) and, if not, it can avoid the need for a system call in the trivial cases of locking and unlocking a lock with no contention. This is illustrated in [Figure 2.1](#), where both threads are able to acquire the lock without contention (assuming that thread one releases the lock before thread two tries to acquire it). Since system calls can cause context switches and are hence expensive, this optimisation makes synchronisation much cheaper in the low-contention case.

The general case when system calls are required is when a futex address is set to the value chosen by the user to denote it being ‘locked’. This requires a trap into the system call (`futex_wait`) to wait for it to be changed. From the perspective of a lock, this would be when a thread possesses the lock and a different thread would like to obtain it. This is illustrated in [Figure 2.2](#), where thread two sleeps for a period of time waiting for the lock to be released by thread one.

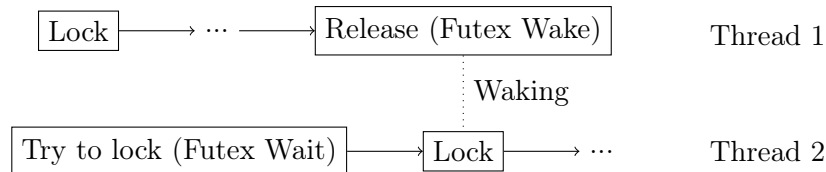


Figure 2.2: Example of a contended lock, showing how four lock actions require only two system calls, which are required due to the contention.

Threading libraries on Linux, such as `pthread`s, internally use the `futex` system call for synchronisation. As a result, by paying attention to the `futex` calls of a multithreaded program and their arguments, a lot can be learnt about how that program is synchronising. GAPP utilises kernel-level tracing [8], and so is a good candidate for extending to experiment with this idea.

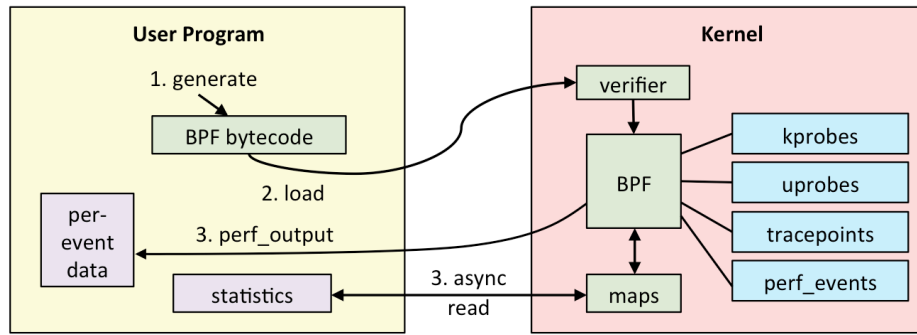


Figure 2.3: Diagram of eBPF internals, from [21]

2.6 eBPF: tracing in the Linux kernel

The Linux kernel itself offers a few options for tracing programs without any source code modification at all. Of most interest for this project is eBPF (extended Berkeley Packet Filter)², which is based around a sandboxed virtual machine running in the kernel that allows the running of user-defined programs efficiently in kernel-space.

An eBPF program is typically composed of a series of event handling functions, which are automatically executed when their corresponding events occur. There are a variety of events, ranging from a kernel function being executed to a TCP packet being received.

It has its origins in BPF, which was created to enable fast in-kernel filtering for packets (hence *packet filter*) for tools like `tcpdump` [22], but which has become more powerful over time [23], and now handles events far removed from its initial domain of network packets. In practice, the terms eBPF and BPF are used interchangeably to refer to eBPF.

A diagram presenting the internals of eBPF is shown in Figure 2.3. BPF bytecode can be written by hand, however, just like machine code, it is usually generated as it is tedious and difficult to write, and there are compilers which generate it from C. As shown in the diagram, the generated bytecode is verified before execution. This verification provides guarantees that collectively aim to ensure that no unsafe code can execute, and is the reason why eBPF code can be safely executed within the kernel context - achieving this without eBPF would require one to write a much more heavyweight kernel module.

For this project, the eBPF data sources of most interest are the events which are triggered upon the entering and leaving of kernel functions, or **kprobes**. These cause a receiver to be called whenever any process enters a specific system call, and the analogous **kretprobes** can call a receiver whenever any kernel function *returns* (one can imagine deriving information such as average syscall latency from a pair of corresponding **kprobes**).

²Julia Evans has an approachable and detailed overview of the wider tracing systems available in Linux at <https://jvns.ca/blog/2017/07/05/linux-tracing-systems/>.

```

1 from bcc import BPF
2 BPF(text="""
3 int kprobe__sys_clone(void *ctx) {
4     bpf_trace_printk("Hello from BPF!\\n");
5     return 0;
6 }
7 """).trace_print()

```

Listing 2.2: A minimal BCC program.

eBPF enables us to hook into these probe points and achieve a wide variety of useful tracing - a relevant example is tracing calls to the `futex` call, which was alluded to in [subsection 2.5.4](#). We encounter some significant limitations of these probes in this project, which are discussed in [subsection 3.3.1](#).

Another source of interest are **uprobes**. These are similar in nature to the kernel probes described above, but attach to a userland function. They span all processes running the given binary containing the function which probes are attached to [24]. An interesting example to demonstrate their utility presented by Brendan Gregg [23] is the ability to see all input to bash by attaching a `uretprobe` to the `readline` symbol in `/bin/bash`.

eBPF programs running inside the kernel would be very limited in use if they were unable to communicate some results back to the user-mode applications that created them. As shown in [Figure 2.3](#), the programs are able to do this in three main ways: through the use of `perf_output`, which allows sending arbitrary data using a custom struct; through BPF maps; and finally using the `bpf_trace_printk` method, which is typically for debugging, and sends a string with basic formatting support along a pipe known as the `trace_pipe`.

BPF maps are key/value data structures which can be accessed from both kernel space and user space. They allow the eBPF program to write data which can then be asynchronously read back by a user front-end that can use the data however it likes. There are a number of abstractions over them, allowing a wider variety of data structures than just simple maps.

2.6.1 BCC

This project will use the BPF Compiler Collection, or BCC, which simplifies the writing of BPF programs by allowing the compilation of C into BPF bytecode, along with supporting a front-end for consuming the data in user-space. The front-end can be written in a number of languages - C(++), Lua and Python. For the purposes of this report we will only look at the Python front-end.

A minimal BCC program is shown in [Listing 2.2](#). It will print *Hello from BPF!* each

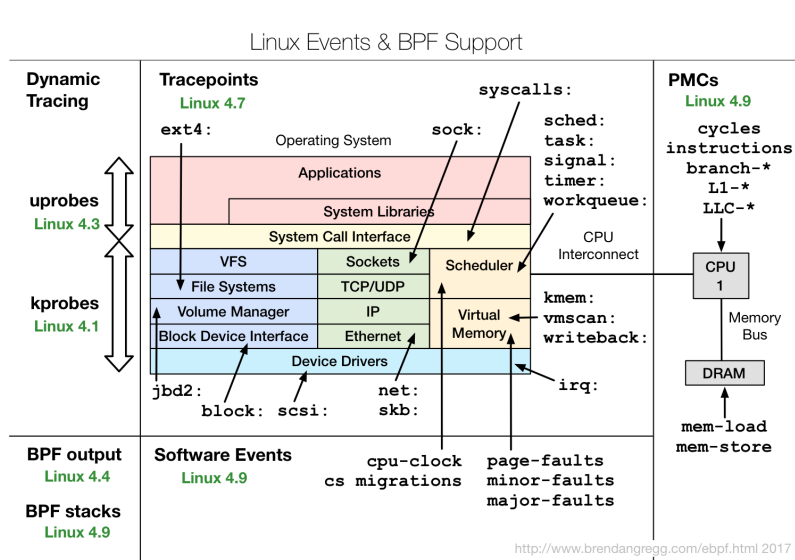


Figure 2.4: Supported events in eBPF by Kernel version. Diagram from [21]

time a new process/thread is started, by hooking into the `sys_clone` system call.

There are a few things going in this example which are worth explaining. Firstly, the Python front-end is responsible for providing the source of the BPF program to compile. It does this through the BPF object, which takes it as a string, `text`. This can, of course, be sourced from e.g. an external file, but for the purposes of brief examples in this report, they are included as a string literal. Another point of note in the Python code is that it calls `trace_print` on the returned object. As mentioned, calls to `bpf_trace_printk` are sent to a pipe. The `trace_print` function essentially blocks, and routes this pipe to standard output, so that the output can be viewed.

Other points of interest in the example, this time in the C code, include that the function name is important - the `kprobe__` prefix signals to BCC to automatically set this up to be a `kprobe` for the system call name denoted by the rest of the function name (`sys_clone` in this case). In real code, this linking is usually done manually by the Python front-end rather than relying on ‘magic’ function names in this manner, but this approach works fine for a demo. Finally, it is necessary to include the return 0 - this value is read by the BPF executor, and some functions have specific behaviour based on the value that is returned, such as to drop packets or redirect them.

2.6.2 Performance of eBPF

An important factor to consider for profiling tools is their run-time overhead. If a performance analysis tool slows your program down substantially³, it is more likely to end

³An exception to this rule is tools which *deliberately* slow down parts of your program, and account for this delay in their calculations.

up affecting the runtime of the program in a way that hides real performance bottlenecks, particularly if certain threads have more tracing overhead than others. Accordingly, the performance impact of using BPF is a concern which should be considered.

The Linux kernel versions which have eBPF with the features important to us are fairly recent - for example, support for `kprobes` was only introduced in Kernel version 4.1 [23] (Figure 2.4 details eBPF features and the relevant Kernel version) which was released in 2015. Because of this, there is not yet a significant body of research covering the overhead which eBPF and these features in combination can induce. Research we did find, though, was positive - finding the overhead of eBPF “not ... significant” [25], and that there was only an 8% reduction in performance attributable to the usage of BPF [26].

Naturally, this overhead depends on to what extent you use eBPF: if you are simply registering an entry probe for a function which is called irregularly, it is reasonable to expect a very low overhead. Conversely, if you register a handler with lots of complex logic to a function that is used regularly, the opposite is true.

3 | Extending GAPP

3.1 Overview

3.1.1 Goal

The high-level goal of this project was to end up with a tool which can take a binary and identify, classify, and detail potential parallelism bottlenecks. A key goal, however, is that it must be able to do this without requiring manual instrumentation of the source code (beyond compiling with general debug information) or depending on a specific threading library.

3.1.2 Implementation

To achieve its goals, this project has been built on top of GAPP [8], which traces the kernel’s scheduling decisions to identify potential bottlenecks. For example, suppose a parallel application with four threads. If one thread acquires a lock, and the other three wait to acquire it, they will get de-scheduled into a non-runnable state. By tracking the quantity of threads in a non-runnable state, and reporting stack traces when threads are scheduled out at these times, it’s possible to identify potential parallelism bottlenecks, and so GAPP reports stack traces to a Python front-end whenever specific conditions based on this data are met (this is further described in [subsection 2.4.4](#)).

The project as a whole is split between a Python front-end and the eBPF back-end. The eBPF side is written in C, which is compiled into eBPF bytecode by the Python side (using BCC) when the tracing application is started.

Since the eBPF backend is event-driven, and does not have any sort of ‘initialisation’ function, any data that needs to be initialised needs to either be a constant baked into the compiled C code, or checked for and initialised at the start of every single event where it is relevant. Additionally, communication at run-time is one-way from the back-end towards the front-end. As a result of this, some parameters are injected into the C source code string dynamically by the front-end before it is compiled, and for this reason it is compiled on demand at runtime.

3.1.3 Feature summary

As previously discussed, GAPP is unable to distinguish between different issues that lead to bottlenecks (like synchronisation, writing files, etc.); it outputs raw stack traces alongside a quantification of how critical they are. This calculation weights how many threads were blocked with how long they were blocked for to arrive at a number denoted as their *criticality metric*. This project extends GAPP to classify identified potential bottlenecks into two broad categories: synchronisation and IO, with IO supporting both network and file interactions. A further category, *unknown*, is used when the system cannot discern a particular cause.

Additionally, bottlenecks are associated with additional data to make them more useful to developers trying to understand them.

Synchronisation stack traces are associated with the corresponding stack traces that either woke them or corresponding to the thread which they woke. For example, when a stack trace corresponding to a thread being de-scheduled to wait upon a lock is reported, the front end will report it alongside the stack traces corresponding to the stack traces of the code which unlocked it.

GAPP merges identical stack traces by adding their criticality metric together. For example, an individual ‘lock’ stack trace may be reported multiple times, and all of these traces are merged for the output into a single stack trace with the sum of the criticality metrics. Throughout these different instances of an individual lock acquisition, the waker stack trace (the ‘unlock’ trace) may vary. As a result, an individual reported synchronisation stack trace may be associated with a number of different stack traces which woke it. To help understand that, this project shows these corresponding stack traces are weighted by the ratio of the amount they occurred ([subsection 3.6.1](#)).

However, this does not give a full story - imagine a list with a global lock. If 95% of the times the lock is acquired is for a ‘get’ method which finishes very promptly, but 5% of the time the lock is acquired for a very slow ‘sort’ operation, which takes a thousand times longer than the ‘get’ method to complete, then it may be reported that, most of the time, threads waiting on the lock are woken by the ‘get’ method completing and unlocking the lock. However, in reality, while the ‘get’ method happens more often, the real bottleneck may be that the ‘sort’ method, despite making up a smaller number of lock releases, causes the other thread to have to wait for far longer to acquire the lock.

To help resolve this issue, this project adds a summary page of all locks (aggregated from all synchronisation activity), and reports them alongside the wakers of the locks, weighting by the criticality of the wakers where possible. This way, a user of the tool can quickly determine which locks are particularly critical in their system by seeing which ones have the highest criticality wakers. This is documented further in [subsection 3.6.2](#).

In addition to this extra synchronisation-related detail, IO stack traces also get extra information reported alongside them, consisting of the file names or IP addresses they were interacting with, where known. This, too, is weighted by the criticality metric.

As well as the above, other key extensions to GAPP that have been developed for this project are summarised as follows:

- A terminal user interface was created to cope with displaying the amount of extra information being produced ([subsection 3.7.3](#)).
- Stack trace outputs have been enhanced to include line numbers and simplify traces of C++ threads ([subsection 3.7.2](#)).
- Processes can be traced by their file name or process identifier instead of just the former ([subsection 3.7.1](#)).
- The tool can spawn a process itself and begin tracing it, rather than requiring the user to run it themselves separately ([subsection 3.7.1](#)).
- The most critical individual ‘unlocking’ actions are reported as part of the summary to help quickly identify the most problematic synchronisation ([subsection 3.6.3](#)).
- The most critical actions on files are reported as part of the summary ([subsection 3.6.4](#)).

3.1.4 Comparison

To complement the discussion above of features added by this project, and the following sections which detail the implementation details of this project, this section briefly discusses the practical differences a user may experience between the original GAPP and the extended version produced by this project.

The summary details of both are shown in [Figure 3.1](#) for the original GAPP and [Figure 3.2](#) for the new tool.

Beyond this, the two versions diverge quite a lot: the original version of GAPP shows some top critical functions and lines (in the same format as those shown in the new version, but omitted for brevity), and then prints out all of the critical stack traces, split by whether they were detected within a library function or whether they were detected in user code.¹

On the other hand, the new version of GAPP moves these stack traces to the ‘all’ page, adding information on the derived causes of the stack traces. The extra details for

¹The extended version of GAPP does not make this distinction, because some of the more important stack traces which are being classified are typically identified as library traces, though it would be straightforward to port this feature.

Criticality Metric for each thread	
8536	126267
8542	170705
8540	183907
8539	184296
8535	187559
8541	191266
8538	197990
8532	221108
8533	222819
8534	238239
8537	240886
8531	395656
8545	547444
8546	6353605
8549	8678636
8547	9099080
8548	9738362
8550	10959693
8551	13919259
8552	16053747
8553	31743836
8555	35618270
8554	35942955
8556	64724868
8530	180706397
Sum =	426646850

Post Processing time in milli seconds: 3

Total switches: 177 Critical switches: 70

Figure 3.1: An example of the summary information of the original version of GAPP from this project. There would also typically be top critical function and lines information, in the same format as in the extended version screenshot.

synchronisation and IO traces are only visible when viewing the corresponding page of the interface, which reduces the noise on the ‘all’ page (the menu for changing page is visible in [Figure 3.2](#)). There is also the lock page, which is further detailed in [subsection 3.6.2](#).

On the summary page, this project adds two new elements, including a list of especially critical synchronisation actions, and a list of especially critical file actions. These are distinct from the overall critical stack trace reporting, and this difference is detailed in [subsection 3.6.3](#) and [subsection 3.6.4](#). More screenshots of the new tool are shown in [Appendix A](#).

3.1.5 Backend summary

To aid in understanding the following sections, a diagram of the backend structure is included in [Figure 3.3](#). It is not expected that the reader fully understand it yet, but instead that it be used as a clarifying reference for the rest of this chapter, and to help clarify what this project contributes versus what GAPP started with.

Notably, the overall structure of the final backend is that various probes set flags and data which are later included as additional metadata in the critical stack traces sent to the front-end by the modified underlying GAPP codebase.

Additionally, various data is sent from some of these backend probes to the frontend,

```

GAPP Results | bin/multi_mutex | 'q' to quit

SUMMARY | ALL | SYNCHRONISATION | IO | LOCKS

73 threads were discovered and tracked.

Criticality Metric per thread:
17284: 2152979511
17195: 1925093022
17261: 1735524246
17302: 1667026873
17262: 1639589845
17193: 1312448277
17304: 1299973384
17299: 1299814198
... and 65 more.
Total: 13032449356

Total switches: 1480
Critical switches: 1404
Post Processing time (ms): 173

Top critical functions and lines, with frequency:

big_work(int) -- 6195
    multi_mutex.cpp:16 -- 4879
    multi_mutex.cpp:15 -- 1311
    multi_mutex.cpp:18 -- 5

medium_work(int) -- 29
    multi_mutex.cpp:27 -- 21
    multi_mutex.cpp:26 -- 8

small_work(int) -- 7
    multi_mutex.cpp:38 -- 6
    multi_mutex.cpp:37 -- 1

```

Figure 3.2: An example of the summary page of the extended version of GAPP from this project.

including filenames and additional stack traces (unrelated to critical stack traces), which are used to provide extra data for the critical stack traces that are displayed.

It is worth noting that GAPP already has thread tracking logic, but this project largely moves it behind an abstraction layer of accessor functions in order to make some of the logic easier to follow and build on top of, so our approach is rather divergent.

Data flowing into the `sched_switch` tracepoint probe means that it is used to influence extra data included in the critical stack trace events sent to the frontend.

3.2 Causation flag system

GAPP [8] identifies potential bottlenecks by analysing the scheduling event which is triggered when a scheduling decision is made (i.e. a process replaces another process on a processor core).

This project identifies potential causes for these changes by adding a flag which is managed for each thread in the back-end, which we refer to as a *causation flag*.

The code for managing these flags is contained within `MGAPP/bpf_data.h`, and contains

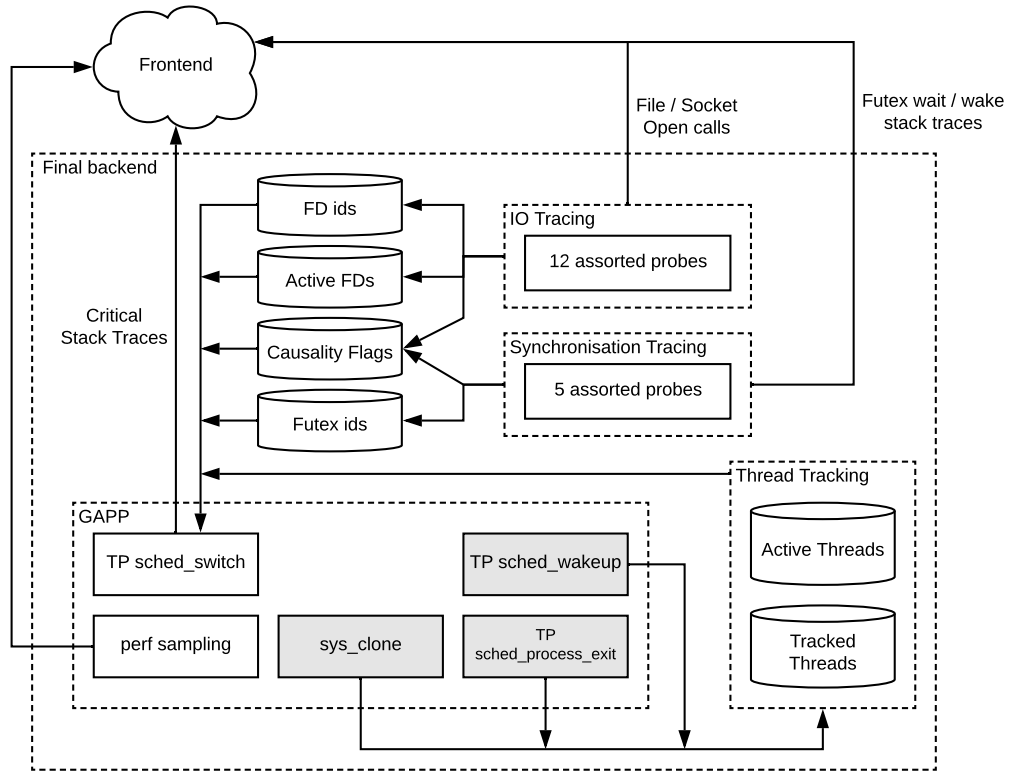


Figure 3.3: A diagram showing a high-level overview of the backend. Probes with grey backgrounds are *return* probes, and TP denotes a tracepoint probe. There is an additional implicit dependency from thread tracking to all probes, since they all check whether the thread that caused the probe is being traced or not.

a BPF hashmap, `last_thread_operation`, which maps thread IDs to their current flag value. There are associated set, get and clear methods, which are used by the rest of the codebase to abstract away dealing with the map directly.

By default, the flag values are zero, denoting that if a stack trace is triggered for a corresponding thread, the ‘cause’ is unknown - that is, the system is not certain whether it corresponds to a synchronisation bottleneck, an IO bottleneck, or just that it was de-scheduled during a critical period by coincidence.

The flags that exist are listed in [Figure 3.4](#). The most interesting point of note is that synchronisation is split into two categories: waking and waiting. This separation is

Flag meaning	Flag value
Unknown	0
Synchronisation (Waiting)	1
Synchronisation (Waking)	2
I/O.	3

Figure 3.4: A list of the flags which may be set for a thread

important to be able to correctly associate the corresponding waker/waiter stack traces discussed earlier (and further in [subsection 3.6.1](#)), but would not be necessary outside of this requirement. Waiting synchronisation refers to when a thread causes a critical stack trace by sleeping, and waking synchronisation refers to when a thread causes a critical stack trace either by being de-scheduled while waking a thread from synchronisation-related sleep, or by switching immediately to it after waking it.

Initially, an alternative approach was considered where file and networking IO are considered separately, however this proved tricky to implement as many Linux system calls, such as `sys_read`, work with any file descriptor, including those corresponding to network sockets. This could have been made to work by tracking which file descriptors correspond to files and which correspond to sockets. We deemed that this was not a hugely useful distinction when it is typically evident from the stack traces what type of IO is going on, as well as the fact that most of the time IP addresses or file names are listed alongside the stack traces due to the extra detail this project attaches, and in the front-end IP addresses and file names are just conflated into values in one dictionary of file descriptor to string.

It is worth noting that the ‘unknown’ state does not necessarily indicate that the classification system is lacking or incorrect - due to how potential stack traces are identified, some stack traces are not themselves relevant to a bottleneck. For example, if most of the program’s threads are wrestling over a lock, every descheduling event will get reported as a critical stack trace. This means that other threads which have nothing to do with the lock can get caught up in the mix when they get descheduled because their time-slice ended. They are not causing any bottleneck, so they cannot be classified into a category like synchronisation or IO. However, in reality, it should usually be more likely for stack traces which can have a cause determined to be reported the most, as they should get reported more regularly than incidental, non-bottleneck-related stack traces. We find this holds true in the evaluation.

3.3 Identifying Synchronisation Bottlenecks

3.3.1 Limitations of return probes

When a kernel function return probe is registered through eBPF, it functions behind the scenes by registering a corresponding entry point probe which performs set-up necessary ready for the return probe to function correctly. This involves caching the current return address of the function and overwriting the return instructions to ‘trampoline’ into the eBPF executor [27]. Since the kernel functions can have multiple concurrent threads executing them, there can be a number of different return addresses which require caching at any given point. For this reason, an in-kernel array of the `kretprobe_instance` struct is

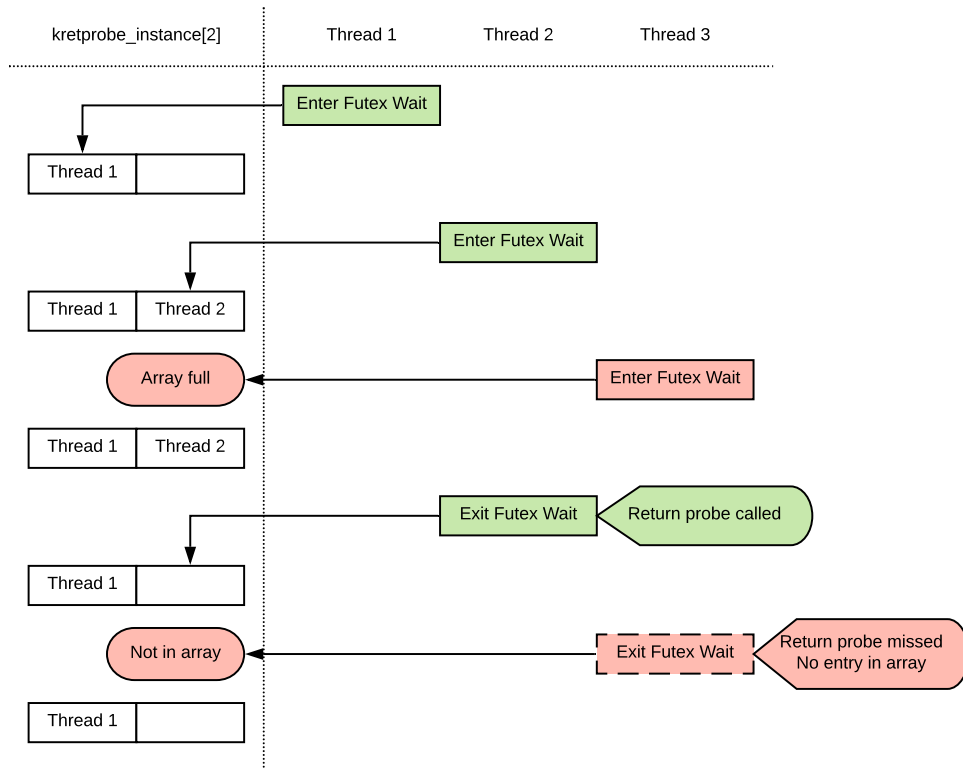


Figure 3.5: A diagram to visualise how a return probe can be missed. In reality the length of the `kretprobe_instance` array is likely to be larger than two elements, but in this example it is small for brevity.

allocated, which consists of entries which each hold information about a currently-executing (including if the thread executing it is currently switched out) invocation of the function.

This number of these objects for each registered return probe is not particularly generous by default; usually set proportionately to the number of CPUs in the machine [27], and this means that if enough threads are concurrently calling the same kernel function, then they can consume the available space for `kretprobe_instances` and be unable to set up the return probe, meaning it is not called.

This is generally not a problem for kernel functions which return promptly and do not block, as it is very unlikely for a sufficient number of threads to get descheduled in the same routine without any of them being rescheduled and exiting it. Most kernel functions meet this criteria, since they are generally brief and operate in constant time without any blocking operations.

For a kernel function that can lead to blocking, however, it is easy for calls to accumulate from different threads or processes. Unfortunately, this means that if you register an entry probe and an exit probe on a kernel function, there is *no guarantee* that the return probe will be called, even when the entry probe has been called and the function returns correctly.

This is illustrated in [Figure 3.5](#), where three threads all trigger an entry probe, but one misses the return probe as there was no space to set up the `kretprobe_instance` when it entered the function.

That this can be a problem was discovered [28] while probing the `sys_futex` system call, and registering both an entry and return probe. Using the arguments passed to the system call, an initial version of the code tracked futex waiters and their corresponding wakers using an entry and exit probe. However, there was a persistent issue where wait and wake calls were being detected, but the corresponding return probes for the wait calls were not always being called. This became a significant issue due to `mysqld` running on the system, which permanently has a large number of threads waiting on a futex with a low timeout (around half a second), and re-entering when they wake. Because of this, when the probes were registered, the first few return probes of the application being traced were being correctly identified, but eventually a sufficient number of MySQL threads had woken and returned to sleep on a futex to permanently consume all of the `kretprobe_instance` structures.

The `kretprobe_instance` structures are local to each individual return probe registered, which means that this issue does not affect the viability of return probe tracing in general, but makes probing the exit points of functions which may block such as `sys_futex` or `sys_write` tricky.

3.3.1.1 Avoiding this issue

Having identified this issue, it becomes important to have a way to avoid it affecting results. For some uses, return probes are still a viable approach (and are still used in some places in this project), but for anything that is likely to block significantly they are not.

Instead, two main approaches have been taken for many parts of this project.

First, when a return probe would be directly associated with a tracepoint or another thread's activity, we trace that instead. For example, instead of probing the return of a futex wait call, we probe the entry points of the *wake* call that wakes it up, and then the tracepoint that corresponds to it being woken up, allowing us to effectively determine when the thread would return from the futex wait call and do the appropriate clean-up and follow-up actions there instead.

If this is not possible, we instead try to *approximate* the return probe. For IO ([section 3.4](#)), we cannot trace the return probe of the close call reliably as it can block for a period. Instead, we store some state when we enter the close function, and additionally trace a short function that the close function calls prior to ending. This is not a perfect solution, since it's possible that things we care about happen between entering that function and the real return from the function, but in practice this is not a major problem, as the

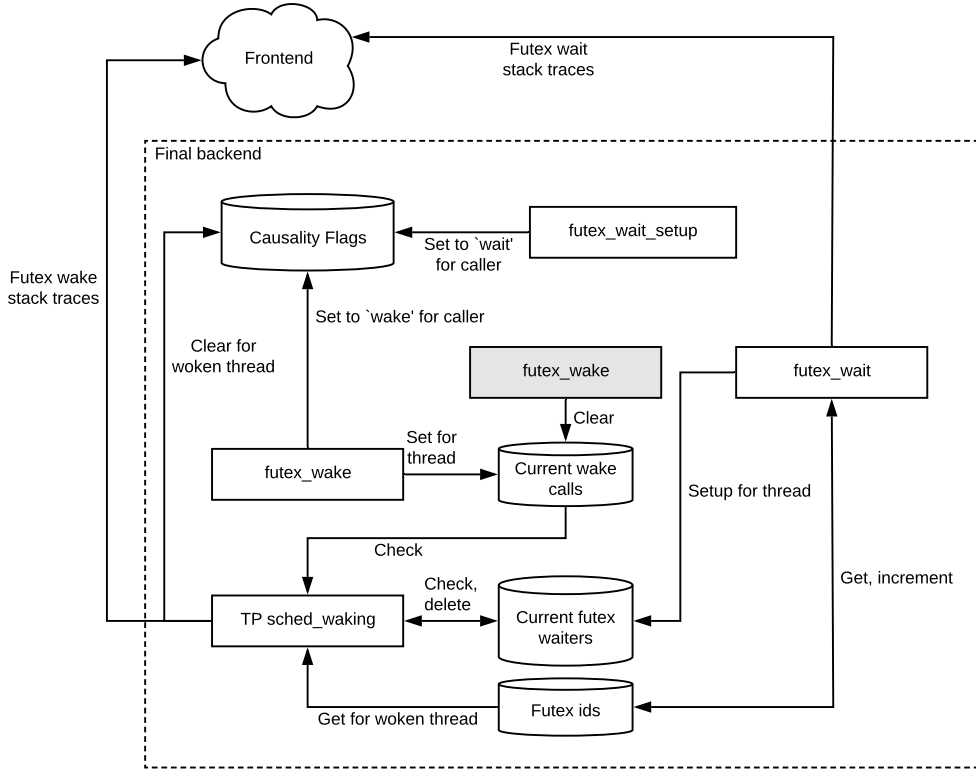


Figure 3.6: A diagram showing a high-level overview of the probes registered for the backend related to synchronisation activity. Probes with grey backgrounds are *return* probes, and TP indicates tracepoint probes.

function probed towards the end has been selected to be suitably short and non-blocking.

3.3.2 Summary

A summary diagram of the backend for synchronisation bottleneck detection is included in Figure 3.6. Again, this is intended more for referencing against while reading the following chapter, rather than being an independent documentation of the software in its own right.

The code relevant to tracking futex wait operations is held within `MGAPP/futex.h`.

There are two scheduling decisions which can be directly associated with a synchronisation bottleneck: those that happen when a thread wanting to acquire a futex gets descheduled, waiting to be woken; and those that happen when a thread wakes up a thread waiting on a futex, causing a new thread to get scheduled.

We additionally give each thread a monotonically increasing *futex id*, which is stored in an eBPF hashmap named `futex_inc_counter_h`. This maps thread identifiers to their

Field name	Purpose	Type
<code>event_id</code>	Futex event identifier for this thread; used in post-processing to associate stack traces related to the same event.	int32
<code>stack_id.</code>	Identifier for a stack trace for this wait event in the <code>futex-stack_traces</code> table.	int64
<code>tgid</code>	Thread group ID, used in stack trace processing.	int32
<code>pid</code>	Thread ID that is going to be waiting.	int32
<code>futex_uaddr</code>	User address for futex. Currently not used, but could be used in future post-processing for uses such as identifying specific contended locks.	int32

Figure 3.7: A list of the fields of the futex wait event which is dispatched to the Python handler for every futex wait.

current counter value, and is accessed in the code via our related `get` and `increment` methods. This counter is included with reported stack-traces and used during post-processing, and it is incremented for a given thread when it begins waiting on a futex, so that both the wait and wake actions relating to that specific futex action can be associated later on during post-processing.

3.3.3 Tracking futex wait operations

When an active thread attempts to acquire a futex but cannot do so without contention, it enters into the system call `sys_futex`, [29, `kernel/futex.h:3683`]. This enters into `do_futex`, which then checks which futex operation is desired and enters `futex_wait`. [29, `kernel/futex.h:2696`]. This performs a number of setup operations before sleeping.

Within `futex_wait`, there are a number of lines of setup prior to a `‘retry’` label (which allows a `goto` statement to move execution to that position). After the `retry` label, there is a call to `futex_wait_setup`. This is important because a wake-up in `futex_wait` may be ‘spurious’. It is not completely clear what this means, or if this explanation is exhaustive, but based on comments in the kernel source, we believe this is related to signals waking up the thread. Under these circumstances, it can return to sleep by jumping to the `retry` label, which re-runs `futex_wait_setup` and waits until the next wake-up.

Two probes are registered for futex wait: one to the `futex_wait` method itself, and one to the `futex_wait_setup`. The former does a few major jobs:

1. Increment the *futex id* count for the thread in question
2. Adds an entry to a BPF map named `futex_waiting_h`, which is used later by wake-related handlers
3. Submits a *wait event* (detailed in [Figure 3.7](#)) to a BPF perf buffer to be picked up by the Python side and processed. This is further discussed in [subsection 3.6.1](#).

The latter probe, on `futex_wait_setup`, sets the causation flag for that thread to denote that it is currently doing synchronisation activity. This flag is set here, rather than in the probe for `futex_wait` because the flag is cleared whenever GAPP dispatches a critical stack (so that it does not falsely associate future scheduling decisions relevant to that thread with scheduling). As a result, if this thread re-enters via the mentioned `retry` label, the flag has been reset to unknown. By setting the flag in this setup method, we ensure that it is always correctly set.

3.3.4 Tracking futex wake operations

The code for tracking futex wake operations is split between `MGAPP/futex.h` and `MGAPP/scheduling.h`, because it makes use of the `sched_waking` tracepoint.

There are three main probes registered:

1. An entry point probe for `futex_wake` [29, `kernel/futex.h:1579`].
2. A return probe for the `futex_wake`.
3. A tracepoint probe for `sched_waking`.

The reason for registering these three probes is that a single futex wake call can wake any number of wait calls (including none). As a result, rather than trying to trace pieces of futex wake individually, we trace the waking scheduling point and sandwich it with futex wake probes.

The futex wake entry probe is used to set the causality flag for the thread to `FUTEX_WAKE`, so that if it is descheduled during a critical period then the stack trace is correctly associated as pertaining to a synchronisation bottleneck.

It also registers two entries in BPF hashmaps. One, in `futex_waking_h`, maps the `uaddr` pointer of the futex that the call is relevant to this thread (the waking thread), and the other maps the current `pid` to the `uaddr` which is used in the return handler.

The return handler merely uses the `pid` to `uaddr` map to delete the entry in the `futex_waking_h` map, as well as clearing the causation flag set on the thread by the entry probe.

The `sched_waking` probe looks up the thread being woken in the `futex_waiting_h` hash to identify whether it is waiting on a futex. If so, the `uaddr` it is waiting on is looked up in the `futex_waking_h` hash. If a match is found, then that means this wakeup is a futex-related wakeup. If this is confirmed, a stack trace is sent to the Python front-end via the `futex_wake_events` BPF perf buffer. The data sent with this is detailed in [Figure 3.8](#).

As well as sending a stack trace, the probe additionally sets up some state in a map called `futex_wake_id_h`. It maps the *waking* thread to the *futex identifier* of the *woken*

Field name	Purpose	Type
<code>event_id</code>	Futex event identifier for this thread; used in post-processing to associate stack traces related to the same event.	int32
<code>stack_id</code>	Identifier for a stack trace for this wait event in the <code>futex_stack_traces</code> table.	int64
<code>tgid</code>	Thread group ID, used in stack trace processing.	int32
<code>waking_pid</code>	Thread ID for the thread that is doing the waking.	int32
<code>woken_pid</code>	Thread ID for the thread that is being woken.	int32
<code>futex_uaddr</code>	User address for futex. Currently not used, but could be used in future post-processing for uses such as identifying specific contended locks.	int32
<code>sleep_time</code>	The time (in nanoseconds) that the waking thread spent sleeping on this futex. Currently unused, but could be used to fuel future post-processing.	int64

Figure 3.8: A list of the fields of the futex wait event which is dispatched to the Python handler for every futex wait.

thread. This is so that if a critical stack trace is sent for the *waking* thread by GAPP, the correct futex identifier corresponding to this action is reported. There is more detail on this in [section 3.5](#)

It is worth again emphasising that the stack traces which are sent that have been detailed in this section are **not** treated in a similar way or conflated with the critical stack traces that GAPP’s core code sends. Their use is detailed in the front-end discussion ([subsection 3.6.1](#)).

3.4 Identifying IO Bottlenecks

Identifying IO bottlenecks is, in principle, more straightforward than synchronisation, as only one thread is involved in the action, and there is no need to correlate activity between a pair of threads as there was for synchronisation. However, in practice, the kernel-level APIs are less helpful for our probes as many accept a `FILE *` pointer, which cannot be mapped to a file descriptor within BPF code. This is due to the fact that the `FILE` struct varies significantly between systems, and the accessor method and struct definition are within `glibc`, which cannot be compiled to BPF. There is thus no trivial portable way to get a file descriptor from this struct.

As a result, some additional logic is needed to trace these functions while also getting hold of the file descriptor to be able to report the relevant file descriptor back to the front-end when sending up critical stack traces related to IO.

Further, a seemingly significant amount of IO activity takes place in `glibc`, within which we trace some functions as a compromise between wanting to have most tracing at

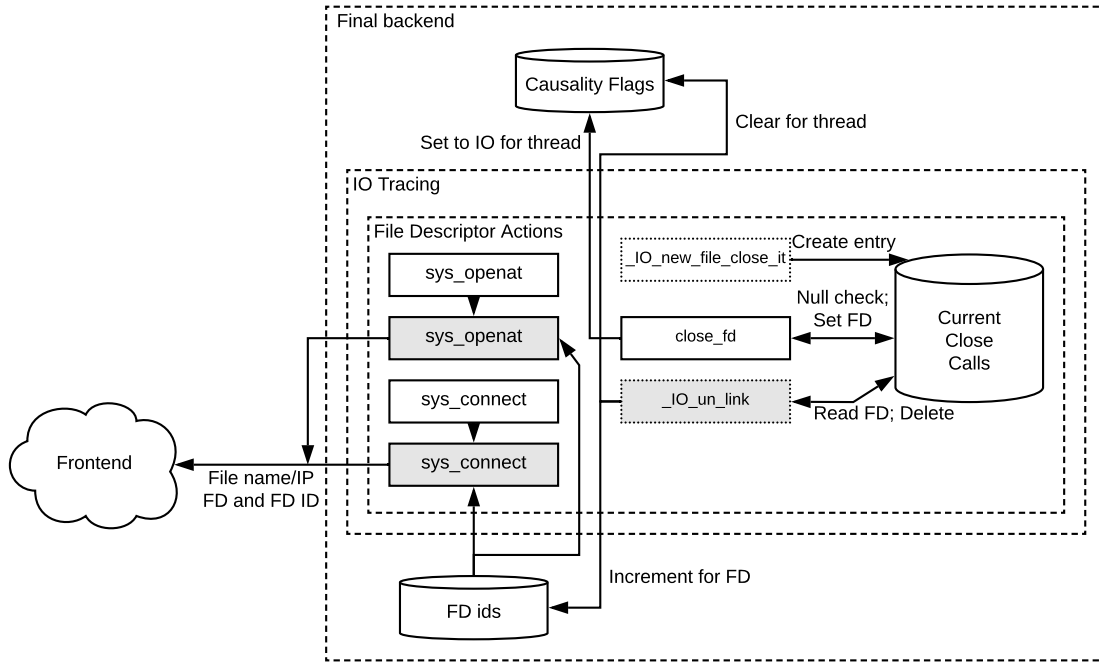


Figure 3.9: A diagram showing a high-level overview of the probes registered for the IO backend related to file descriptor tracking. Probes with grey backgrounds are *return* probes, and a dotted border denotes a *glibc* function being probed.

a kernel level (to reduce the number of assumptions about client program dependencies) but also wanting to maximise utility of our tool. Fortunately, most binaries and runtimes end up going through this, not just C. For example, C++ also links to *glibc*, and most interpreted language runtimes route through it at a lower level.

3.4.1 Identifying IO Bottlenecks - Files

One of the difficulties in tracking file operations is the desire to report the name(s) of the files causing the bottlenecks, rather than just the stack traces where they were discovered. This is complicated by the fact that a file descriptor number can be reused when a file is closed and a new one is opened.

Our need to avoid taking any risks with breaking the memory limitations of the BPF virtual machine mean that these file names cannot be stored in any persistent BPF memory. Instead, they are be sent to the Python front-end as soon as possible. Unfortunately, we are not aware of any ordering guarantees on different BPF perf buffers, and so it is uncertain whether the open events would be processed by the Python front-end in the correct order

relative to actions on the file descriptor.

For example, assume that a file is opened, read from (causing a critical stack trace to be reported), closed (ergo the file descriptor is no longer used, and can be reallocated), and then later a new file is opened, and given the same file descriptor number as the old file. Without ordering guarantees, it is conceivable that both open actions might be processed by the Python front-end before the critical stack trace is processed. This means that it is incorrectly associated with the *second* file name.

To avoid this, a unique incrementing number we call a *file descriptor identifier* is associated with each file descriptor number within a BPF hash map named `fd_ids`. Every time a file is closed, the file descriptor number's entry in this table is incremented so that the next file that gets assigned that file descriptor has a different associated number. There is additionally a second BPF map which tracks the *active* file descriptors for each thread - that is, when a thread is doing IO, this map is set up to map the thread identifier to the file descriptor it is doing work on. These two numbers (the file descriptor and the corresponding file descriptor identifier) are reported alongside all file-related critical stack traces sent to the Python front-end by GAPP so that, even without any ordering guarantees, they can be associated with the correct file name

Closing files is trickier, because some of the blocking takes place within `glibc` instead of the kernel. Close is prone to blocking because writing to a file is generally buffered until the file is closed, when it is actually synced to disk (or, at least, a lower-level buffer). We found from initial experimentation that tracing just `sys_close` failed to classify all stack traces corresponding to a file close, while the approach we developed to trace the `glibc` function is able to reliably classify them.

The difficulty that arises is that the `glibc` function takes a `FILE *` pointer, not a file descriptor, which is of little use to us for reasons explained at the start of this section. To get around this, we trace the entry to the `glibc` function *and* the entry to the system call for closing files. When the `glibc` function is entered, we set up a dummy entry in the active file descriptor table to denote that a close call is ongoing. Then, when the system call probe is triggered, we override the dummy value with the file descriptor. The system call is engaged almost immediately from the call to the `glibc` function, so it is extremely unlikely the thread will get descheduled before this file descriptor is set up. We further set the causation flag for the thread to denote that it is currently doing IO in this probe.

This is not the only element of complexity to the close system call, however - the likelihood of long blocks means that we cannot rely on a return probe. Instead, we analysed the `close` function within `glibc` and found that just prior to returning it runs a function named `_IO_un_link`, which is very brief and non-blocking. We trace the return probe of this function, and use it as a near-enough approximation of the return time of the close call - cleaning up state, unsetting the thread's active file descriptor, and unsetting the

causation flag for that thread.

3.4.2 Identifying IO Bottlenecks - Networking

Networking is a complicated subject, and there is a wide area which would be valuable for a tool like this to cover. For example, if a program requests a website, first a DNS query is resolved to get the IP address, and then a socket is established to send the HTTP request and download it.

For the purposes of this project, we are primarily interested in achieving a proof-of-concept for classifying networking IO traces - demonstrating that it is possible to classify and detail these bottlenecks using kernel tracing, and so we are particularly interested in that communication over the socket.

Sockets, like files, are identified by a file descriptor on Linux. When a socket is connected (which we trace using a probe on `sys_connect`), we send up an event to the front-end containing the socket's file descriptor, our file descriptor identifier, and the IP address it is connected to.

Socket closing goes through the same routine as for closing files, and so we do not have any specialised logic for that; it is shared.

3.4.3 Identifying IO Bottlenecks - Reads and Writes

It is fortunate for this project that sockets on Linux are abstracted over by a specialised file descriptor. This means that probe code for reading and writing can be shared for all IO, and needs not make a distinction between socket and file IO.

For this, we trace a number of probes, illustrated in [Figure 3.10](#). Each of the entry probes sets the causality flag, sets the active file descriptor for the current thread to the file descriptor being read/written from/to, and sets up some state for the exit probe to work with. The exit probe then clears the causality flag.

We found that the `read` system call often missed return probes, and ended up tracing the `sys_exit_read` tracepoint instead. Ideally, we would have migrated the whole read/write functionality to use the tracepoints for entering and exiting the read and write system calls, as the tracepoints are more reliable than return probes, but our current implementation is satisfactory and there was not a strong enough case to justify the time switching to another implementation and carefully verifying it still works correctly.

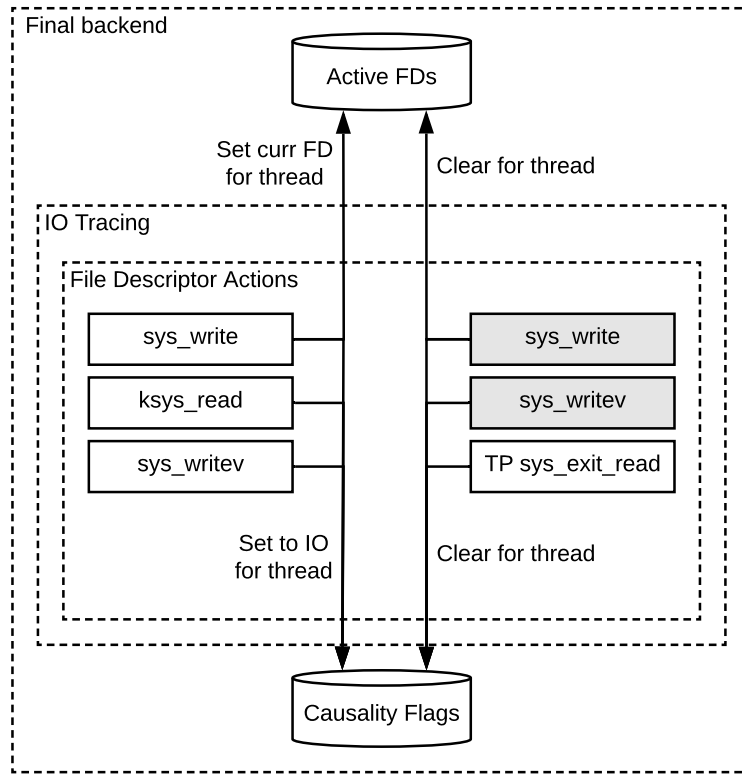


Figure 3.10: A diagram showing a high-level overview of the probes registered for the IO backend related to read and write activity. Probes with grey backgrounds are *return* probes, and TP indicates tracepoint probes.

3.5 Modifications to the core GAPP algorithm

The previous sections discuss how new probes are set up to send additional information to the front-end about synchronisation activity and IO activity, and how the *causality flags* are maintained for each thread.

However, it remains to be discussed how this information actually gets associated with actual critical stack traces reported by GAPP - the primary focus of this project, after all, was to classify these critical stack traces.

This specific modification is relatively straight-forward. Since all of the metadata about traces is stored in BPF maps, ranging from the causality flags to the futex and file descriptor identifiers², all that remains is to include this metadata with the critical stack traces which are sent.

²as a reminder, these latter two ‘identifiers’ are monotonically increasing numbers associated with a thread and file descriptor respectively that uniquely identify a specific pair of futex activities (wait and the corresponding wake) and instances of file descriptors (since different files can share the same file descriptor at different points in time)

The GAPP algorithm for sending these stack traces has been modified to add this data, and adds three new fields to the structure GAPP sends. One stores the causality flag for the relevant thread, denoting the cause the backend has determined for it, while the other two are variable ‘metadata’ entries which vary based on the causality flag.

For an unknown stack trace, the extra metadata entries are unused. For synchronisation stack traces, the first is set to the futex identifier of the thread that is waiting or has been woken, and the second is unused. For IO, the first is set to the active file descriptor, and the second is set to the current file descriptor identifier for the active file descriptor.

3.6 Python front-end processing

Events reported from the eBPF components of the project are being continuously polled for and processed by the Python front-end.

This data is received and processed in a variety of ways, which is explored in the following sections.

3.6.1 Additional synchronisation stack traces

Recall [section 3.3](#), about the backend mechanisms for detecting synchronisation traces. Every pair of futex wait and wake calls (that is, the wait call, and then the corresponding wake call that wakes it up later on) have a stack trace reported with a futex identifier alongside the thread identifier of the thread that waits, and the user-space memory address of the futex. This $(pid, futex_id)$ tuple forms a unique identifier for that futex activity pair.

The corresponding stack traces are stored with the futex address in a front-end dictionary `futex_events`, indexed by this key. Since critical stack traces (i.e. those that would have been reported by the original GAPP implementation) related to synchronisation have this primary key encoded with them in the metadata fields (as discussed in [section 3.5](#)), the corresponding wake/wait call can later be found by looking its key up in the `futex_events` table.

This table will grow at a rapid rate for applications which have a significant amount of futex activity. It would be possible to garbage collect this table by aggregating together data for the lock page ([subsection 3.6.2](#)) and other uses in advance with periodic runs of an algorithm to do so. This project does not contribute this algorithm due to time constraints, but it would certainly be feasible as a future extension.

3.6.2 Lock page

By using the aggregated futex event data, we can produce quite interesting and helpful statistics about overall lock usage. While the critical synchronisation stack traces portray some information about locks, it can be easy for a single lock to cause dozens of stack traces to be output, and it can be a matter of signal to noise if you have some critical locks which are not fixable causing dozens of critical stack traces to be output.

As a post-processing step, this project utilises the `futex_event` table, and creates a mapping from futex address (i.e. lock) to the corresponding stack traces. We reason that the wake stack traces tend to be the most useful since, when they have a corresponding critical stack trace, the criticality of that trace can be indicative of how significant the action was in terms of monopolising use of the lock.

We then split these wake traces into two categories: those with associated critical traces, and those without. For those with associated critical trace, we sum up the criticality metrics and merge identical stack traces together under one unified sum (similarly to how GAPP does for standard critical stack traces). For those without, we just merge identical ones together.

Between those with and without associated critical stack traces we then have a complete list of unique stack traces which ‘unlocked’ this futex. We then order these locks by the criticality of the most critical *individual* unlock stack trace (not the aggregated sums³) and display them on the lock page of the program. We list each lock, and within each lock’s section we print their critical wakers in order, and then finally their wakers with no criticality associated. An example of the output is included in [Figure 3.11](#).

This information is collated as part of post-processing, and contributes to the post-processing time. The algorithm does iterations over a space of all critical stack traces and all futex events, so is $O(nm)$ in complexity, but this could be reduced by pre-processing the critical stack traces into a map of futex keys to critical stacks in advance, which would make it $O(n)$ in the number of futex events. This was not done for this project due to time constraints, and the fact that post-processing time remains fast.

3.6.3 Most Critical Futex Activity

The summary page has been enhanced to show the top five individual most critical traces which were reported. These don’t provide anywhere near the same value or utility as the lock page or the enhanced synchronisation stack traces can, but they can provide a

³The reason we sorted by individual stack trace is because a lock with millions of very low criticality calls is likely harder to fix than a lock with few very high criticality calls. The sum of criticality for the former may be higher, but the latter will have higher individual criticalities. An alternative strategy could be to use an average, but we have not taken this approach yet.

```

GAPP Results | bin/multi_mutex | 'q' to quit

SUMMARY | ALL | SYNCHRONISATION | IO | LOCKS

Lock (uaddr 6316352) (Top criticality: 284789590):

Most critical unlockers:

Criticality 6690236563:
  __lll_unlock_wake
    <--- std::mutex::unlock() at std_mutex.h:122
    <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
    <--- big_work(int) at multi_mutex.cpp:21
    <--- random_work(int) at multi_mutex.cpp:46
    <--- << C++ Thread Initialisation >>

Unlockers with no criticality (no particular order):

__lll_unlock_wake
  <--- std::mutex::unlock() at std_mutex.h:122
  <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
  <--- medium_work(int) at multi_mutex.cpp:32
  <--- random_work(int) at multi_mutex.cpp:51
  <--- << C++ Thread Initialisation >>

__lll_unlock_wake
  <--- std::mutex::unlock() at std_mutex.h:122
  <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
  <--- small_work(int) at multi_mutex.cpp:43
  <--- random_work(int) at multi_mutex.cpp:49
  <--- << C++ Thread Initialisation >>

```

Figure 3.11: An example of the lock output for a program with a single lock accessed from three places.

quick at-a-glance look at where the worst synchronisation bottlenecks in your application are likely to be.

Again, these are based on individual reports, rather than a sum of all identical stack traces as in the GAPP-style stack trace reporting.

3.6.4 Most Critical File Activity

Similarly to [subsection 3.6.4](#), the summary page also has up to five entries about critical file activities detected. Again, these are individual rather than an aggregate sum of identical stack traces, in order to provide an at-a-glance view of the worst offenders in your application.

3.7 User Experience

This section focuses on changes focused on providing a better user experience for users of the extended GAPP tool.

3.7.1 Tracing relevant threads

One of the challenges of writing code to be executed in the eBPF virtual machine is identifying what invocations of probes that you care about: an eBPF probe gets called from every process on the system that calls the relevant function, even kernel workers. This is useful for programs which trace, for example, all networking behaviour on a system, but less useful for tracking the behaviour of a single application, which we do. Since most events are irrelevant to the application being traced, it is important to be able to quickly determine whether this is the case and terminate the probe's execution early in these cases.

GAPP solved this issue by checking the first few letters of the `comm` (a string representing the name of the executable) of the thread that caused the probe to run. This approach has two main drawbacks:

1. If multiple instances of the program are running at the same time, they are not distinguished and the results reported will be produced by the union of the probes of the two processes, which is likely to be undesirable. Worse, other processes with the same prefix may get traced and their results conflated.
2. An instance of GAPP has to be running in one terminal instance while the user executes their program in another terminal. This impacts usability, but is not individually a major issue.

To solve this, this project can be invoked with a number of command line arguments. Firstly, the user must provide either `path` or `pid` as a positional argument. Depending on their choice, different lines are injected into the project's C code before execution.

This system works by having all probes use a single function (`tracking_thread_should_be_tracked`) to check whether a thread should be being traced, and the definition of this method is changed using `#ifdef` macros. For example, the Python code can inject a line `#define PROG_PID <pid>` at the beginning of the code before compilation to ensure the version of the function which checks the pid is used.

For all methods of tracing, every probe will check if the relevant thread they are being called about should be being traced (using the method mentioned above), and if so they will add it to a BPF map (`thread_pids`) of traced threads if it isn't already in it.

Then, any thread which is started from a thread being traced (through `sys_clone`) is noted and added to the same map, to ensure that if the initial thread is correctly traced then all threads of that application will be traced.

An additional method of tracing was added for convenience, which is used by using the `path` argument and passing the flag `-s`, short for *spawn*. This method spawns a process based on the path to the binary provided, and traces the relevant pid.

As well as being more convenient, this solution is actually somewhat necessary: tracing by name alone means that multiple instances of the program will get conflated, and tracing by pid alone means you need to have already started running the program to get the pid before you can start tracing. While a best-effort approach is made to pick up on existing threads that aren't being traced and begin tracing them if they appear to be part of the application, this is not guaranteed to pick up on anything and long-sleeping threads, for example, can influence what stack traces are considered critical (and hence reported).

When using the spawn parameter, a short C++ program which was made for this project named *bootstrap* is used, which is invoked with the path to a binary and a list of arguments. After a few seconds of waiting, it uses the `execvp` function to switch to executing the desired program without losing its pid. During the waiting period, the eBPF probes are attached and the tracer boots up, so that by the time it has switched, every action of the program can be correctly traced and monitored.

3.7.2 Enhanced Stack Trace Reporting

Report stack traces with BCC involves sending a stack trace from the eBPF backend to the Python frontend (actually creating the stack trace is handled in the backend by the BCC method `get_stackid`), and then iterating over the reported stack trace in the frontend, which is in the form of a list of addresses.

```
big_work(int)
  <--- random_work(int)
  <--- void std::__invoke_impl<void, void (*)(int), int>(std::__invoke_other, void
d (&&)(int), int&&)
  <--- std::__invoke_result<void (*)(int), int>::type std::__invoke<void (*)(int)
, int>(void (&&)(int), int&&)
  <--- decltype (__invoke((_S_declval<0ul>)(), (_S_declval<1ul>)())) std::thread:
:_Invoker<std::tuple<void (*)(int), int> >::_M_invoke<0ul, 1ul>(std::_Index_tuple<0ul,
1ul>)
  <--- std::thread::_Invoker<std::tuple<void (*)(int), int> >::operator()()
  <--- std::thread::_State_impl<std::thread::_Invoker<std::tuple<void (*)(int), i
nt> >::_M_run()
```

Figure 3.12: An example of a stack trace with GAPP's default processing.

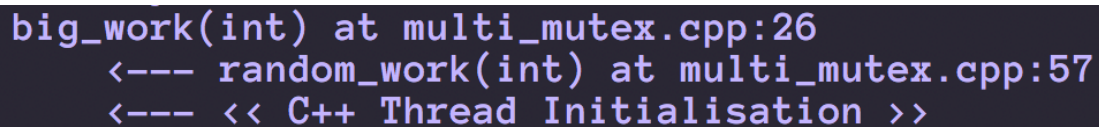
GAPP uses two techniques to translate addresses from BPF stack traces into code locations. For critical stack traces, it uses BCC's `sym` function which takes an address and returns the corresponding function label in the binary.

For the critical functions and lines feature, however, it uses the `addr2line` utility to translate the addresses. This utility provides more information, including source-code line numbers, provided debug information is baked into the client binary, but is somewhat less reliable - the exact circumstances are unclear (potentially due to relying more on debug information which isn't present in linked libraries), but there are occasions when BPF's `sym` function is able to translate an address, while `addr2line` returns a series of question marks to denote its inability to translate the address.

This project introduces a flag, `-e`, which enabled 'enhanced' stack trace reporting. With this enabled, instead of translating stack trace addresses with `sym`, they are first converted into a hex form and then translated with `addr2line`. Only if that fails for an individual address does the tool gracefully fall back onto using `sym`.

Further, many 'pieces' of stack traces are commonplace and contribute nothing but noise to the user. For example, when creating a thread through C++'s standard library, all created threads end up with five additional, dense lines full of C++ template type notation and obscure standard library types.

To combat this, when enhanced stack traces are enabled, a table of common patterns is traversed. This table contains a series of pairs of data: one element of the pair is a list of expected stack trace lines, and the other is the string to replace it with when detected.



```
big_work(int) at multi_mutex.cpp:26
  ---- random_work(int) at multi_mutex.cpp:57
  <-- << C++ Thread Initialisation >>
```

Figure 3.13: An example of a stack trace with the enhanced processing enabled. File names and line numbers are added, and the C++ thread initialisation boilerplate is simplified.

The stack traces are then processed line-by-line for each pattern, and any incidents are replaced. Currently, this is done for a number of C++ thread initialisation patterns, and additionally for some boilerplate TBB ([section 2.3](#)) traces. It is trivial to add new patterns in future, due to how this feature is implemented - they can just be added to the table of common patterns.

3.7.3 User Interface

A major usability issue that developed over the course of the project was a growing information dump into the console. GAPP's method of output was to print a summary of detected critical functions and lines, then the critical stack traces, and finally some extra information such as the number of switches and how long post-processing took.

This approach breaks down with the amount of extra information that is added by

this project - in particular, the extra stack traces displayed with synchronisation traces to show their wakers. Displaying this information in a single dump of text is possible - but far from ideal. It becomes necessary to wade through lots of data to get to any desired information if the user has a particular requirement (such as if they are looking specifically for synchronisation bottlenecks).

As a result, a dynamic user interface was designed using Python bindings for the *ncurses* library, which allows terminal interfaces to be developed with relative⁴ ease (particularly compared to writing code such as portable console buffering manually).

The interface is broken up into pages: a summary page, showing information such as functions which have been identified as particularly critical by GAPP's rankings, a page showing all reported stack traces, along with what this project believes the cause is (or unknown), but has no additional information such as the wakers for synchronisation traces, the file names for IO traces, etc. There is then a page for each type of bottleneck, which lists all of the extra information associated with them, and finally the lock page described in [subsection 3.6.2](#).

3.8 Engineering larger systems with BCC and eBPF

An interesting side effect of this project was gaining insight into engineering larger systems using BCC - as it is a relatively new technology, the ecosystem surrounding it is not very developed, and best practices for developing larger projects are unclear.

3.8.1 Separation of code

Due to BCC still being a maturing technology, tooling which we often take for granted when using other platforms is either very new or simply doesn't exist, and best practices are still being developed. This is exacerbated by the fact that it has a relatively small developer community, by virtue of how low-level it is and how specific its uses are.

For example, a typical way which most of the existing tools and examples [\[30\]](#) using BCC are written is to embed the C portion of the application within a Python multi-line string. This works well for smaller programs, which most BCC programs are - typically, BCC programs are single-file applications focused on a specific issue; Execsnoop [\[31\]](#), for example, traces new processes created with the 'exec' system call - but begins to break down as the programs get larger. This is for a variety of reasons, including:

- Development environments currently do not provide support for nesting C within a

⁴strong emphasise on relative - we had to manually implement scrolling, pagination, and creating boxes of the right size involved ugly manual buffering code

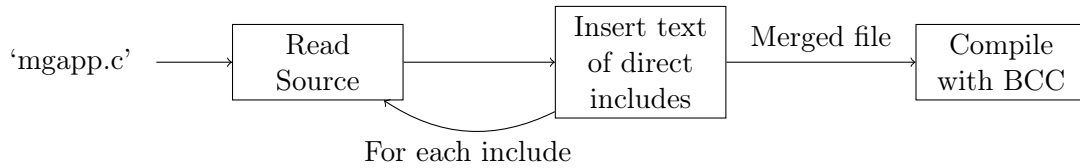


Figure 3.14: Process of generating the backend file for compilation. Direct includes are processed before compilation and the include line is replaced with the contents of the file. This process is done recursively.

Python string like this, meaning no syntax highlighting is available, no completions can be done, automatic refactoring is impossible, etc.

- Separate compilation is difficult to achieve, which means linting and pre-compilation are unavailable, lengthening development times and requiring regular mental context switches between development and executing within a terminal to ensure correctness of the changed code.
- As a program grows and its logic becomes more complicated, the file grows and it becomes difficult to separate modules of the codebase in a meaningful way.

The original GAPP codebase was around 400 lines of C code embedded in a string, with around 300 lines of Python code following it. This isn't unreasonable, but as this project grew around it, it soon started to become impractical to maintain and extend.

The most obvious way of improving this situation is extracting the C code into a separate file and loading it in Python to compile it. This mitigates many of the tooling issues, since C tooling such as IDEs and linters can assist with editing.

However, this is still not sufficient: the final code archive for this project has around 1,300 lines of C code, and this is too much to reasonably include in a single file. Unfortunately, this cannot be solved trivially: BCC does processing to code *prior* to the C preprocessor running, and C files are including through the use of the `#include` preprocessor directive. This means that if one places eBPF-related code (such as a map lookup) within a file that is included using `#include`, then these calls are not translated into the correct bytecode and thus fail, since the BCC substitutions for such methods occur *before* the file is included and do not re-run afterwards. This makes this method of including files work correctly for 'ordinary' files such as Linux include headers, which are not specifically intended to be used for eBPF programs, but not for files made for this project.

Pursuing a method of separating BCC C code into multiple files has a number of requirements. Primarily, it remains important to be able to include files at a specific spot in the code, since C is sensitive to the order of the definition of methods, variables and data types. Additionally, it would be beneficial to ensure that IDEs can still understand the code separation to provide auto-completion and basic linting support.

In the end, a basic line-based parsing system was developed in our frontend where files are *seemingly* included as usual, with `#include`, however a comment is added at the end of the line to denote that this include needs to be processed *prior* to compilation (and hence the preprocessor execution). Lines of the form `'#include "<file name>" //DIRECT'` are replaced with the contents of the referenced file by the Python front-end. Code editors, however, do not recognise any importance in the comment and simply treat them as a normal include, allowing their completion systems and code understanding to function correctly. This process is illustrated in [Figure 3.14](#)

Another, smaller issue, is that eBPF constructs such as map definitions are not defined in terms of C macros (even though they look like they may be at a glance). As a result, code editors do not understand them and their lexers and parsing trees are unable to understand C files featuring these constructs. To combat this, a second case was added, for includes that should be *removed* before compilation (of the form `'#include "<file name>" //REMOVE'`), and a file was created with fake definitions for these constructs, such as `'#define BPF_PERCPU_ARRAY(A, B, C)'`. Again, this approach allows editors to comprehend and assist with development of eBPF-related code, but still allows compilation, since the import is removed before actually compiling the code.

One problem that is created by this approach is that C compilation errors become incredibly unhelpful. While the semantics of a C `include` are essentially equivalent to copy-paste dumping the text of the included file, there is a major difference: the compiler is aware of this, and accounts for it when reporting an error. If you include a file which is ten lines long, and have a syntax error a few lines further down, you will still have the correct (pre-include) line number reported - not offset by ten lines due to the file inclusion. Unfortunately, as this project subverts the compiler's file inclusion process, the compiler loses this information and you get incorrect line numbers (and file names, if the syntax error is in another file) reporting, as these compilation errors are reported respective to their position in the single file which results from concatenating all of the various include definitions together.

This, too, can be mitigated, however. By using the `#line <file name> <number>` macro, the compiler can be directed to act as if the current line is what's in the macro. By injecting a `#line <file> 1` prior to each direct include, generated compiler errors correspond correctly to the file name and line number they originated in.

By utilising these tricks, the project's code has been successfully split across a number of files, without compromising on the ability of IDEs and tooling to understand it as valid C code nor the ability for the C compiler to generate helpful errors.


```

1 static inline int is_null(int id) {
2     int *ptr = ... // obtain pointer
3     return (ptr != NULL)
4 }
5
6 int a_function() {
7     bool is_null_a = is_null(0);
8     bool is_null_b = is_null(1);
9     if (is_null_a && is_null_b) {
10         ...
11     }
12 }

```

Listing 3.1: Example code to generate a post-optimisation *or* operation on pointers.

3.8.2 Debugging issues

Using BCC, C code is compiled using Clang into LLVM, an intermediate language for compilers, before being compiled into BPF bytecode and finally being verified by the kernel’s verifier before execution.

While the C compiler itself is good at producing useful error messages, the eBPF bytecode verifier is not so helpful: errors are typically cryptic to the point of having almost no value unless you are handwriting the bytecode, and those that aren’t as cryptic are often still vague and not particularly helpful.

For example, eBPF bytecode cannot represent the logical *or* operation on two pointer types (with the expected semantics of determining whether either pointer is non-null). A specific issue relevant to this that came up in development is illustrated in [Listing 3.1](#). This entailed the usage of a method which returned an *int* representing whether a pointer is NULL or not - 0 if so, 1 if not. Two calls were made to this method, and the results used within an if statement to test whether they are both null. A minimal code example of what this might look like is shown in [Listing 3.1](#)

Potentially contrary to expectations, the C compiler BCC uses (Clang) was doing optimisations which, while sensible and valid, cause this to generate incorrect eBPF bytecode. It first inlines the function and then, rather than *oring* two null checks, it optimises these individual checks away and simply checks whether the *or* of the two pointers is non-zero, as demonstrated in [Listing 3.2](#).

Whilst this optimisation does not maintain the types which operations are applied to, it is typically a safe optimisation to make as the preservation of types at machine code level is generally unnecessary. Unfortunately, since the eBPF runtime has a higher-level notion of types than a processor does, and does not support the logical *or* operation on pointers, this resulted in a cryptic error, which does not reference any position in the C code, but instead an address in the generated bytecode, which is pictured in [Figure 3.15](#).

```

1 int a_function() {
2     int *ptr_a = ...;
3     int *ptr_b = ...;
4     if (!(ptr_a || ptr_b)) {
5         ...
6     }
7 }

```

Listing 3.2: Theoretical example of what a compiler-optimised version of [Listing 3.1](#) may look like, when represented in C.

Due to the cryptic nature of these errors - since the output is usually bytecode which does not directly map to the source code due to optimisations and BCC's rewrites - a significant challenge has been to track down the causes, and typically involved removing pieces of code at a time until it compiles, then attempting to find a work-around which generates bytecode which passes verification successfully.

```

0: (b7) r2 = 0
1: (63) *(u32 *)(r10 -20) = r2
2: (7b) *(u64 *)(r10 -32) = r2
3: (61) r6 = *(u32 *)(r1 +56)
4: (7b) *(u64 *)(r10 -136) = r1
5: (61) r1 = *(u32 *)(r1 +24)
6: (63) *(u32 *)(r10 -36) = r1
7: (63) *(u32 *)(r10 -128) = r1
8: (18) r1 = 0xffff980f103fcc00
10: (bf) r2 = r10
11: (07) r2 += -128
12: (85) call bpf_map_lookup_elem#1
13: (bf) r9 = r0
14: (63) *(u32 *)(r10 -128) = r6
15: (18) r1 = 0xffff980f103fcc00
17: (bf) r2 = r10
18: (07) r2 += -128
19: (85) call bpf_map_lookup_elem#1
20: (bf) r8 = r0
21: (b7) r7 = 1
22: (15) if r8 == 0x0 goto pc+1
    R0=map_value(id=0,off=0,ks=4,vs=4,imm=0) R6=inv(id=0,umax_value=4294967295,
    _null(id=1,off=0,ks=4,vs=4,imm=0) R10=fp0,call_-1 fp-24=0 fp-32=0 fp-136=ctx
23: (b7) r7 = 0
24: (bf) r1 = r9
25: (4f) r1 |= r8
R1 pointer |= pointer prohibited

```

Figure 3.15: An example of an error generated from incorrect eBPF bytecode compiled from valid C. This particular error is an instance of the issue illustrated in [Listing 3.1](#).

This was solved in the project by making one of the two return values `volatile`. While a slight misuse of the keyword in this situation, it ensures the compiler doesn't try to optimise that value away.

4 | Evaluation

4.1 Setup

All of the experiments in this evaluation have, unless stated otherwise, been conducted within a VirtualBox virtual machine running on a host MacOS machine. The virtual machine is running Ubuntu 18.04, using a patched version of Kernel 4.17.0-rc6. It has 8GB of RAM allocated, and 4 virtual cores running at a base clock of 2.9GHz and a boost clock of 4.2GHz. It is running from an SSD, however disk bandwidth has been capped to 10MB/s within the VM (though due to OS and library-level file caching, as well as the access speed bottlenecking not being 100% reliable, generally higher speeds than this are observed by user code).

Outside of the virtual machine, a reasonably standard user workload was running (music player, web browser, text editors). There is generally insignificant CPU load, and the host CPU has six cores total so the four virtual CPU cores should not have been influenced significantly by this. Within the virtual machine, a reasonably clean install of Ubuntu is running, with the only notable difference being an idle MySQL server in the background.

In the code snippets, boilerplate such as imports have been omitted for brevity; they are intended to assist with understanding the evaluation, rather than being valid, compiling pieces of code. Namespace accessors (such as `std::`) are also omitted in places to avoid line breaking and assist with brevity.

Example code has been compiled with optimisations disabled, as they are generally trivial examples which the compiler attempted to delete entirely.

All examples have been run with the `-e` flag to enable enhanced stack trace reporting (discussed in [subsection 3.7.2](#)).

4.2 Note on glibc stack traces and parent threads

For a period, we were unable to succeed in compiling the `glibc` library with the frame pointer enabled due to time constraints and difficulty with building the whole library. The frame pointer is used to unroll the stack traces at program runtime, and as a result, in some of our evaluation, stack traces reported from within the `glibc` have generally failed to unroll fully and do not extend back to the user code that called them.

We later managed to resolve this problem after it become more detrimental during the Parsec benchmark evaluation, and evaluation from that point onwards has, wherever

```

1 void mutex_work(int id) {
2     static std::mutex work_mutex;
3     std::lock_guard<std::mutex> l(work_mutex);
4     std::this_thread::sleep_for(chrono::milliseconds(250));
5 }
6
7 int main() {
8     std::vector<std::thread> threads{};
9
10    for (int t = 0; t < 12; t++) {
11        threads.emplace_back(mutex_work, t);
12    }
13
14    for (auto &thread : threads) thread.join();
15    return 0;
16 }

```

Listing 4.1: `mutex`, an example program for testing basic synchronisation bottleneck detection

necessary, been done against a rebuilt `glibc` with `-fno-omit-frame-pointers -g` added to the `CFLAGS`. The parts of the evaluation prior to this have not been re-done, as it was already clear what the stack traces would have been due to how trivial the programs were (i.e. only one line in the program causing a `write` call).

In addition to this, critical stack traces related to the parent threads were not reported until the same benchmark, due to a minor mistake when enabling parent thread reporting. Again, these previous evaluations were not re-done because the parent threads in the contrived examples are not doing anything of particular value besides waiting on a thread join, which our other evaluations demonstrate is detected correctly.

4.3 Individual feature evaluation

The initial evaluation of the project is focused on ensuring that each non-trivial individual feature works as expected, and is based on relatively trivial code snippets.

4.3.1 Elementary synchronisation

`mutex` (Listing 4.1) is a simple C++ program that launches twelve threads which each race to acquire a mutex and sleep for a period. As a result, there is a trivial synchronisation bottleneck on the acquisition of the shared mutex.

Running `sudo ./mgapp.py -e path -s bin/mutex` (sudo allows for the use of eBPF probes, and the `-e` flag enables enhanced stack trace processing - subsection 3.7.2), the synchronisation bottleneck is detected as expected, as well as an *unknown* stack trace for

GAPP Results bin/mutex 'q' to quit	
SUMMARY ALL SYNCHRONISATION IO LOCKS	
Critical Path 1 (Criticality Metric: 172469):	Cause: UNKNOWN
nanosleep	
Critical Path 2 (Criticality Metric: 131315):	Cause: SYNCHRONISATION
<pre> __lll_lock_wait <--- std::mutex::lock() at std_mutex.h:103 <--- std::lock_guard<std::mutex>::lock_guard(std::mutex&) at std_mutex.h:162 <--- mutex_work(int) at mutex.cpp:14 <--- << C++ Thread Initialisation >> </pre>	

Figure 4.1: All stack traces reported for an execution of `mutex`.

the sleeping action. This is expected, as the sleep itself is not part of our classification system. Since this function is within `glibc`, this stack trace is brief and does not extend into the user code as discussed in [section 4.2](#).

In addition, the summary page’s display of the wakers of the most critical futex waits ([subsection 3.6.3](#)) outputs only the unlock method on the lock, as expected.

The results shown in [Figure 4.1](#) and [Figure 4.2](#) demonstrate that this simple synchronisation is detected correctly by the futex probes correctly, and additionally the single lock is correctly identified as shown in [Figure 4.3](#).

GAPP Results bin/mutex 'q' to quit				
SUMMARY ALL SYNCHRONISATION IO LOCKS				
Critical Path 1 (Criticality Metric: 131315):			Cause: SYNCHRONISATION	
<pre>__lll_lock_wait <--- std::mutex::lock() at std_mutex.h:103 <--- std::lock_guard<std::mutex>::lock_guard(std::mutex&) at std_mutex.h:162 <--- mutex_work(int) at mutex.cpp:14 <--- << C++ Thread Initialisation >></pre>				
Waker (100%):				
<pre>__lll_unlock_wake <--- std::mutex::unlock() at std_mutex.h:122 <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168 <--- mutex_work(int) at mutex.cpp:15 <--- << C++ Thread Initialisation >></pre>				

Figure 4.2: The synchronisation trace reported for an execution of `mutex`.

GAPP Results bin/mutex 'q' to quit				
SUMMARY	ALL	SYNCHRONISATION	IO	LOCKS
Lock (uaddr 6312256) (No associated criticality):				
Unlockers with no criticality (no particular order):				
<pre> __lll_unlock_wake <--- std::mutex::unlock() at std_mutex.h:122 <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168 <--- mutex_work(int) at mutex.cpp:15 <--- << C++ Thread Initialisation >> </pre>				

Figure 4.3: The locks page for `mutex`.

4.3.2 Shared locks

This test looks into how well the system copes with a lock shared between multiple functions, i.e. one with multiple lock and unlock points. In particular, we are interested in identifying whether the different unlock points are correctly identified within synchronisation stack trace’s additional associated traces and summarised correctly on the locks page. Finally, we expect the most critical wakers to be a landslide for the `big_work` function, since it does by far the most work while holding the lock.

`multi_mutex`, shown in [Listing 4.2](#), involves eight iterations of eighteen threads executing `random_work`. This random work method iterates ten times, each time randomly choosing between executing `small_`, `medium_` or `large_work`. These work functions do nothing of value, but simulate real computation by doing some calculations, using a `volatile` variable to ensure the compiler doesn’t try to optimise anything away.

```
Most critical futex wakes:

Criticality 288691102:
    big_work(int) at multi_mutex.cpp:16
    <--- random_work(int) at multi_mutex.cpp:46
    <--- << C++ Thread Initialisation >>

Criticality 287936001:
    big_work(int) at multi_mutex.cpp:16
    <--- random_work(int) at multi_mutex.cpp:46
    <--- << C++ Thread Initialisation >>

Criticality 286789959:
    big_work(int) at multi_mutex.cpp:15
    <--- random_work(int) at multi_mutex.cpp:46
    <--- << C++ Thread Initialisation >>

Criticality 285401136:
    big_work(int) at multi_mutex.cpp:15
    <--- random_work(int) at multi_mutex.cpp:46
    <--- << C++ Thread Initialisation >>

Criticality 282080917:
    big_work(int) at multi_mutex.cpp:16
    <--- random_work(int) at multi_mutex.cpp:46
    <--- << C++ Thread Initialisation >>
```

Figure 4.4: The most critical futex wakes for `multi_mutex`.

We found that, as displayed in [Figure 4.4](#), all five of the most critical wakers reported on the summary page were related to `big_work`, as expected. This does demonstrate a problem in the tool: having five of the same stack trace adds little value, and it would be more valuable to somehow summarise this information. For reasons as discussed previously, we opted not to merge these for this particular part of the information display, but it may be better to omit repeated stack trace and indicate the sum next to the highest individual trace, or provide an average instead.

On the other hand, seeing five stack traces of the same method does an effective job

```

1 static std::mutex work_mutex;
2
3 void __attribute__((noinline)) big_work(int id) {
4     std::lock_guard<std::mutex> l(work_mutex);
5     volatile int x = 5;
6     for (int i = 0; i < id * 10000000; i++) {
7         // computation involving x
8     }
9 }
10
11 void medium_work(int id); // as big_work, but * 25000 in loop
12 void small_work(int id);  // as big_work, but * 5000 in loop
13
14 void random_work(int id) {
15     for (int i = 0; i < 10; i++) {
16         int r = rand() % 100;
17         if (r < 33) small_work(id);
18         else if (r < 66) medium_work(id);
19         else big_work(id);
20     }
21 }
22
23 int main() {
24     for (int t = 0; t < 8; t++) {
25         std::vector<std::thread> threads;
26         for (int i = 0; i < 18; i++)
27             threads.emplace_back(random_work, i);
28         for (auto &thread : threads) thread.join();
29     }
30     return 0;
31 }

```

Listing 4.2: multi_mutex, an example program for testing more sophisticated synchronisation bottleneck detection

of emphasising where the main synchronisation issue is centred, and on that basis the tool does the job satisfactorily. In a real-world, less contrived application, we might expect to see a number of different stack traces as the tail latencies for different methods are often rarely encountered, meaning the most critical stack trace in the program is unlikely to be repeating itself over and over in such an artificial pattern.

In [Figure 4.5](#) we see the first reported synchronisation stack trace with the additional information associated. There is a small number of samples in the critical functions and lines area which are from the other functions, but this is likely because towards the end of execution the threads were able to run from `small_work` or `medium_work` into `big_work` without descheduling (due to lower lock contention when most threads are finished execution).

Finally, in [Figure 4.6](#) we see the reported locks page. As expected, it has identified the only lock successfully, and identified that the `big_work` function is a critical waker. It did

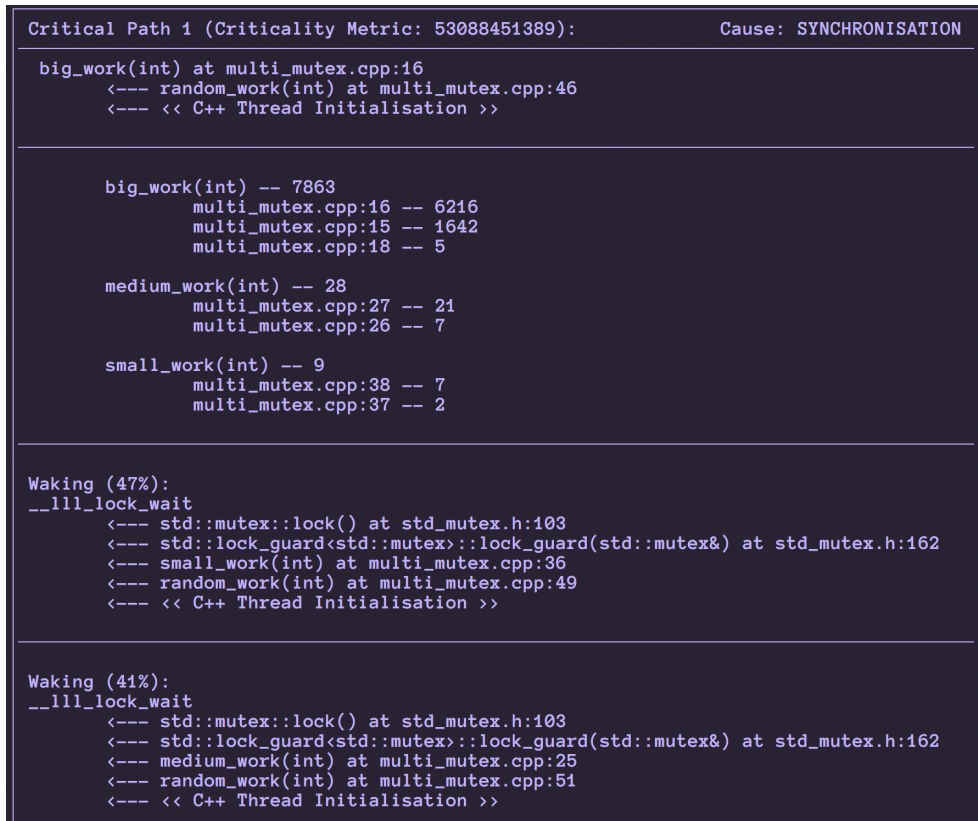


Figure 4.5: The most critical synchronisation trace for `multi_mutex`. Not visible is a third waking trace from `big_work`.

not find any critical waking traces corresponding to the medium and small work functions – this is a notable deficiency of our strategy that has no obvious solution. The problem is that critical stack traces were indeed reported for these unlock points, but were either not flagged during the period of time when the threads are classed as doing `FUTEX_WAKE` activity by our classification system, or had slightly different stack traces. Additionally, wake-related critical stack traces are reported far less commonly than wait-related ones, since a wait event generally directly causes a thread to be descheduled (causing a critical stack trace), whereas a wake-related one doesn't, and we must rely on an element of chance.

4.3.3 Multiple locks

`multi_lock` is a similar program to `multi_mutex` ([subsection 4.3.2](#)), but contain two locks: one lock shared with `big_work` and `medium_work` and one with `medium_work` and `small_work`. This means that we expect a chain of contention from `small_work` to `big_work` as the former should regularly be waiting on `medium_work`, which is in turn waiting on `big_work`. The code is included in [Listing B.1](#) for completeness.

The primary feature we are interested in testing here is ensuring the lock analysis page correctly identifies both locks and reports their wakers correctly.

```

GAPP Results | bin/multi_mutex | 'q' to quit

SUMMARY | ALL | SYNCHRONISATION | IO | LOCKS

Lock (uaddr 6316352) (Top criticality: 249231903):

Most critical unlockers:
Criticality 9807794566:
  __lll_unlock_wake
  <---- std::mutex::unlock() at std_mutex.h:122
  <---- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
  <---- big_work(int) at multi_mutex.cpp:21
  <---- random_work(int) at multi_mutex.cpp:46
  <---- << C++ Thread Initialisation >>

Unlockers with no criticality (no particular order):

__lll_unlock_wake
  <---- std::mutex::unlock() at std_mutex.h:122
  <---- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
  <---- medium_work(int) at multi_mutex.cpp:32
  <---- random_work(int) at multi_mutex.cpp:51
  <---- << C++ Thread Initialisation >>

__lll_unlock_wake
  <---- std::mutex::unlock() at std_mutex.h:122
  <---- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
  <---- small_work(int) at multi_mutex.cpp:43
  <---- random_work(int) at multi_mutex.cpp:49
  <---- << C++ Thread Initialisation >>

```

Figure 4.6: The lock page for multi_mutex.

As shown in Figure 4.7, the tool works exactly as expected for this scenario. Two individual locks are detected, one corresponding to big- and medium- work functions and one the medium- and small- work functions. Similarly, critical wakers are identified for both, and both correspond to the functions we expect to be the critical wakers.

For this particular program, were it not so obviously contrived, this insight could prove very useful - in the real world our functions are typically not named `big_work`, and we don't have such an obvious grasp of their relative runtime complexity - as it is directly telling us which lock is the most problematic and which particular function that uses that lock is the problem.

```

SUMMARY | ALL | SYNCHRONISATION | IO | LOCKS

```

```

Lock (uaddr 6316416) (Top criticality: 366377511):

Most critical unlockers:

Criticality 6919760393:
  __lll_unlock_wake
  <--- std::mutex::unlock() at std_mutex.h:122
  <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
  <--- big_work(int) at multi_lock.cpp:22
  <--- random_work(int) at multi_lock.cpp:48
  <--- << C++ Thread Initialisation >>

```

```

Unlockers with no criticality (no particular order):

__lll_unlock_wake
  <--- std::mutex::unlock() at std_mutex.h:122
  <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
  <--- medium_work(int) at multi_lock.cpp:25
  <--- random_work(int) at multi_lock.cpp:53
  <--- << C++ Thread Initialisation >>

```

```

Lock (uaddr 6316352) (Top criticality: 960697):

Most critical unlockers:

Criticality 2315037:
  __lll_unlock_wake
  <--- std::mutex::unlock() at std_mutex.h:122
  <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
  <--- medium_work(int) at multi_lock.cpp:34
  <--- random_work(int) at multi_lock.cpp:53
  <--- << C++ Thread Initialisation >>

```

```

Unlockers with no criticality (no particular order):

__lll_unlock_wake
  <--- std::mutex::unlock() at std_mutex.h:122
  <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
  <--- small_work(int) at multi_lock.cpp:45
  <--- random_work(int) at multi_lock.cpp:51

```

Figure 4.7: The lock page for multi_locks. The last line of the bottom stack trace is omitted as it would not fit in a single screenshot. It is just a C++ thread initialisation line.

4.3.4 IO - File reading

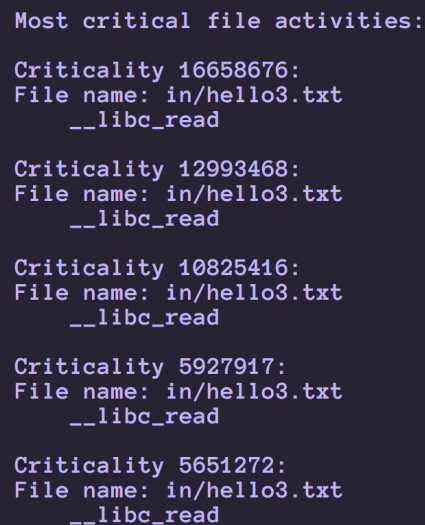
The contrived program `main_read`, shown in [Listing 4.3](#), has four iterations of eight threads simultaneously attempting to each read one of four large (250MB) files (named `hello{1,2,3,4}.txt`). The expectation is that an IO bottleneck should be detected, as all of the threads will be blocking for the read around the same time, because the disk speed is not fast enough to satisfy them promptly. We additionally do not expect any other bottlenecks.

The input files have been created using the command `head -c 250M </dev/urandom >in/hello{1,2,3,4}.txt`, which populates each of the files with 250 MB of random data. Additionally, the command `echo 1 > /proc/sys/vm/drop_caches` was executed as the superuser to avoid the file contents being cached from previous runs of programs which use

the input files.

```
1 void __attribute__((noinline)) big_work(int id) {
2     std::ifstream file("in/hello"
3         + std::to_string((id % 4) + 1) + ".txt");
4     char *buffer = new char[250 * 1024 * 1024];
5     file.read(buffer, 250 * 1024 * 1024);
6     file.close();
7 }
8
9 int main() {
10     for (int t = 0; t < 4; t++) {
11         std::vector<std::thread> threads;
12         for (int i = 0; i < 8; i++)
13             threads.emplace_back(big_work, i);
14         for (auto &thread : threads) thread.join();
15     }
16     return 0;
17 }
```

Listing 4.3: main_read, an example program for testing read IO detection



```
Most critical file activities:

Criticality 16658676:
File name: in/hello3.txt
__libc_read

Criticality 12993468:
File name: in/hello3.txt
__libc_read

Criticality 10825416:
File name: in/hello3.txt
__libc_read

Criticality 5927917:
File name: in/hello3.txt
__libc_read

Criticality 5651272:
File name: in/hello3.txt
__libc_read
```

Figure 4.8: The most critical file activities on the summary page for main_read.

GAPP Results bin/main_read 'q' to quit	
SUMMARY ALL SYNCHRONISATION IO LOCKS	
Critical Path 1 (Criticality Metric: 2755525808):	Cause: IO
__libc_read	
Critical Path 2 (Criticality Metric: 58479007):	Cause: UNKNOWN
std::__basic_file<char>::is_open() const <--- << C++ Thread Initialisation >>	
Critical Path 3 (Criticality Metric: 1312442):	Cause: UNKNOWN
__GI__libc_free	
Critical Path 4 (Criticality Metric: 547669):	Cause: UNKNOWN
std::__basic_file<char>::xsgetn(char*, long)	

Figure 4.9: The ‘all’ page for main_read.

SUMMARY ALL SYNCHRONISATION IO LOCKS	
Critical Path 1 (Criticality Metric: 2755525808):	Cause: IO
__libc_read	
Top files (with criticalities):	
in/hello3.txt: 303656728 31865239	in/hello1.txt: 41279439
in/hello2.txt:	

Figure 4.10: The ‘io’ page for main_read.

Looking at the results in [Figure 4.8](#), we see that the reported most-critical file activities are all related to `hello3.txt`. That they were for this specific file appears to be largely coincidence: different runs gave similar results but for a different file, sometimes with four traces for one file and one for another. It is unknown why one file tends to dominate this metric so much. Our best guess is that it is read first, and therefore has the longest time delta between reads, which may mean less of the file is retained in levels of the cache. Alternatively, it may be starting the read last, and therefore when fewer threads are active - as a result, its criticality score would be weighted much higher.

In terms of the reported stack traces we note that, on the page listing all stack traces ([Figure 4.9](#)), only one stack trace is classified. This initially seems like a disappointing result, but in fact this stack trace has a criticality two magnitudes greater than the others, and is therefore by far the most important stack trace that was detected by the tool. Additionally, the other stack traces largely seem to not be parallelism bottlenecks, so it is correct to classify them as unknown.

Finally, we note that the IO page ([Figure 4.10](#)) successfully identifies three out of four

of the files as causing critical stack traces. That the fourth file is not detected here is due to it not causing any critical stack traces to be reported; we speculate that on this execution the fourth file was read from first, so all threads were running when the relevant threads got descheduled, and therefore GAPP did not class these reads as causing any critical descheduling.

4.3.5 IO - File writing

`main_write` is another contrived program, very similar to `main_read`. It is listed in [Listing B.2](#). The core difference between the two is implicit in the name - it writes files rather than reading them. The expectation is similar to for read; we expect an IO bottleneck to be detected for writing the file. Another difference of note is that the file size written varies - each thread is given an incrementing identifier, and the file size written is proportional to this.

```
Most critical file activities:
Criticality 225875314:
File name: out/7.txt
  __GI___close_nocancel

Criticality 89697736:
File name: out/4.txt
  __GI___close_nocancel

Criticality 83378627:
File name: out/4.txt
  __GI___close_nocancel

Criticality 76769017:
File name: out/7.txt
  __GI___close_nocancel

Criticality 74843997:
File name: out/6.txt
  __GI___close_nocancel
```

Figure 4.11: The most critical file activities on the summary page for `main_write`.

GAPP Results bin/main_write 'q' to quit	
SUMMARY ALL SYNCHRONISATION IO LOCKS	
Critical Path 1 (Criticality Metric: 928094618):	Cause: IO
__GI___close_nocancel	
<pre>big_work(int) -- 5 main_write.cpp:13 -- 5</pre>	
Critical Path 2 (Criticality Metric: 379917210):	Cause: UNKNOWN
__GI___writev	
<pre>big_work(int) -- 1 main_write.cpp:13 -- 1</pre>	
Critical Path 3 (Criticality Metric: 326862787):	Cause: UNKNOWN
__memcpy_avx_unaligned_erms	
<pre>big_work(int) -- 1 main_write.cpp:13 -- 1</pre>	

Figure 4.12: The ‘all’ page for main_write.

SUMMARY ALL SYNCHRONISATION IO LOCKS	
Critical Path 1 (Criticality Metric: 928094618):	Cause: IO
__GI___close_nocancel	
<pre>big_work(int) -- 5 main_write.cpp:13 -- 5</pre>	
Top files (with criticalities):	
out/7.txt: 396148612	out/4.txt: 181962115
out/3.txt: 62334447	out/5.txt: 54762557
out/6.txt: 168974834	out/2.txt: 20728164

Figure 4.13: The ‘io’ page for main_write.

We see in [Figure 4.11](#) that multiple different file write are detected as the top critical writes. This is desirable, since it diversifies the information shown to the end user and makes clear that the problem is spread across multiple files.

In [Figure 4.12](#), we see that only one stack trace was classified as IO, a close call - and that it has the highest criticality. This is because the buffer is only flushed to disk (or, at least, a more permanent cache than the initial buffer it is copied to) when the file is closed.

However, we can clearly see a write call which is unfortunately not classified as IO. This is likely to do with the fact that it was doing work outside of the kernel when the stack traces were reported, and this project currently only traces writes within the kernel.

However, as discussed in [section 3.4](#) for `close`, it is possible to move these probes upwards into `glibc` for a wider surface area to classify traces. We do not have any reason to believe that we could not do a similar technique for the `glibc` write call, as we did for `close`, which would solve this issue.

Finally, we note that in the IO page ([Figure 4.13](#)) many files - six of eight - cause critical traces to flag up. We also note that the two missing files, `0.txt` and `1.txt` would both be the shortest (as the file contents is generated proportionally to the name of the file in the code), and in fact `0.txt` would contain no text. As a result, it is not surprising that these two files do not cause issue, and we claim that this is actually a helpful feature: the files listed are the most critical, and this output very much helps identify the worst offending files.

4.4 Alternative threading library

One of the most critical features of GAPP is that it is independent of the underlying threading library used. This section investigates whether the new features we have implemented are able to work accurately with an alternative library to `pthread`.

4.4.1 Ad-hoc threading

`mutex_man_thread` is a short program functionally identical to `mutex`, presented earlier in [Listing 4.1](#), but which creates twelve threads in an ad-hoc fashion by using the `clone` call rather than using C++'s threads (which themselves used `pthread`). We have no reason to suspect the tool would perform any worse than on `mutex`, however we felt there was value in verifying this fact.

```
1 #define STACK_SIZE 4096
2 static std::mutex work_mutex;
3
4 int mutex_work(void *) {
5     std::lock_guard<std::mutex> l(work_mutex);
6     std::this_thread::sleep_for(chrono::milliseconds(250));
7     return 0;
8 }
9
10 int main() {
11     for (int t = 0; t < 12; t++) {
12         char *child_stack = (char*) malloc(STACK_SIZE);
13         clone(&mutex_work, child_stack+STACK_SIZE,
14             CLONE_SIGHAND | CLONE_FS | CLONE_VM
15             | CLONE_FILES | CLONE_THREAD, 0);
16     }
```



```

17
18     std::this_thread::sleep_for(chrono::milliseconds(10000));
19     return 0;
20 }

```

Listing 4.4: `mutex_man_thread`, an example program for testing ad-hoc thread detection

Indeed, as expected, and as displayed in [Figure 4.14](#) and [Figure 4.15](#), the synchronisation bottleneck and lock are both still detected correctly.

GAPP Results bin/mutex_man_thread 'q' to quit	
SUMMARY ALL SYNCHRONISATION IO LOCKS	
Critical Path 1 (Criticality Metric: 157872): Cause: UNKNOWN	
<pre> nanosleep <--- mutex_work(void*) at mutex_man_thread.cpp:17 <--- clone </pre>	
Critical Path 2 (Criticality Metric: 95220): Cause: SYNCHRONISATION	
<pre> __lll_lock_wait <--- std::mutex::lock() at std_mutex.h:103 <--- std::lock_guard<std::mutex>::lock_guard(std::mutex&) at std_mutex.h:162 <--- mutex_work(void*) at mutex_man_thread.cpp:16 <--- clone </pre>	

Figure 4.14: The ‘all’ page for `mutex_man_thread`.

GAPP Results bin/mutex_man_thread 'q' to quit	
SUMMARY ALL SYNCHRONISATION IO LOCKS	
Lock (uaddr 6299872) (No associated criticality):	
Unlockers with no criticality (no particular order):	
<pre> __lll_unlock_wake <--- std::mutex::unlock() at std_mutex.h:122 <--- std::lock_guard<std::mutex>::~lock_guard() at std_mutex.h:168 <--- mutex_work(void*) at mutex_man_thread.cpp:17 <--- clone </pre>	

Figure 4.15: The ‘lock’ page for `mutex_man_thread`.

4.4.2 TBB

More interesting than ad-hoc threading is TBB ([section 2.3](#)) - Intel’s task-based parallelism library. `tbb_mutex` ([Listing 4.5](#)) is a short program which gives 100 tasks to TBB to execute, each of which acquires a shared mutex and does some work. There is a clear bottleneck in the task which causes execution to serialise along the lock, and we are expecting that to be found to be able to say with any certainty that the tool can handle TBB programs.

```

1 int main() {
2     tbb::mutex work_mutex;
3     tbb::parallel_for(0u, 100u, [&](unsigned n) {
4         work_mutex.lock();
5         int count = 0;
6         for (unsigned i = 0; i < 100000000; i++) {
7             if (n > i || i % 10000 == 0) {
8                 count++;
9             }
10        }
11        work_mutex.unlock();
12    });
13    return 0;
14 }

```

Listing 4.5: `tbb_mutex`, an example program for testing our tool’s compatibility with TBB

```

GAPP Results | bin/tbb_mutex | 'q' to quit

SUMMARY | ALL | SYNCHRONISATION | IO | LOCKS

4 threads were discovered and tracked.

Criticality Metric per thread:
12060: 14497439532
12063: 3226601222
12061: 974019176
12062: 574798011

Total: 19272857941

Total switches: 263
Critical switches: 153
Post Processing time (ms): 13

Top critical functions and lines, with frequency:
main::{lambda(unsigned int)#1}::operator()(unsigned int) const -- 319
tbb_mutex.cpp:19 -- 231
tbb_mutex.cpp:18 -- 88

```

Figure 4.16: The ‘summmmary’ page for `tbb_mutex`.

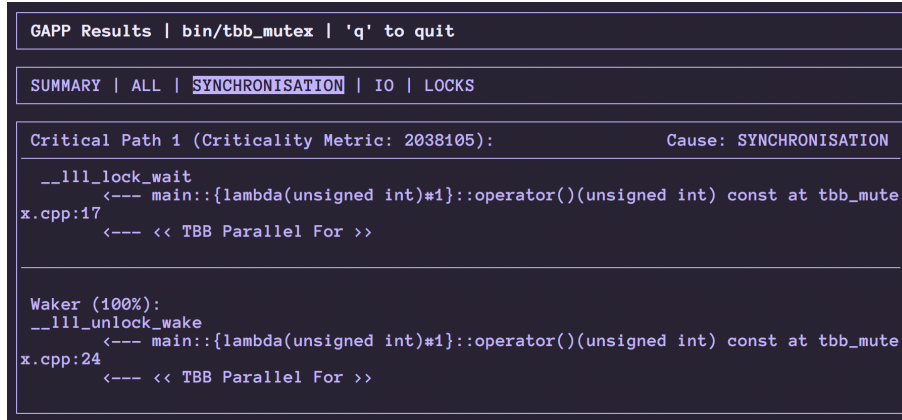


Figure 4.17: The ‘synchronisation’ page for `tbb_mutex`.

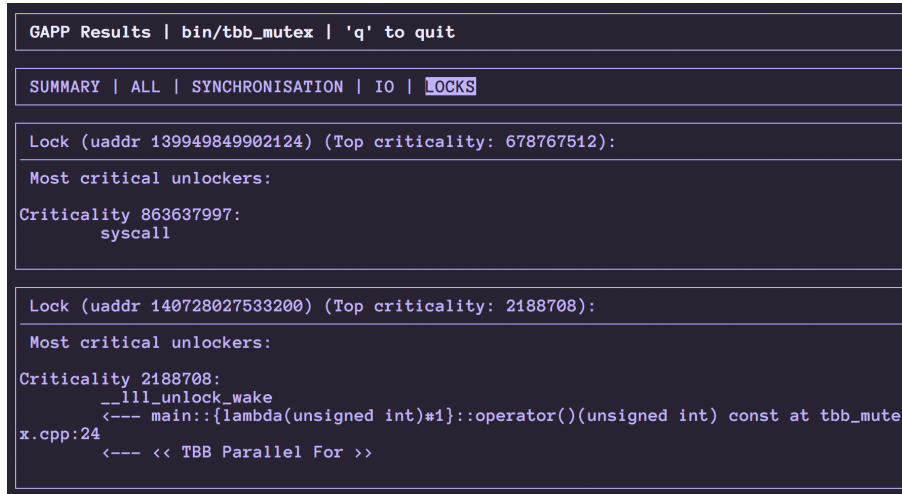


Figure 4.18: The ‘locks’ page for `tbb_mutex`.

As shown in [Figure 4.16](#), [Figure 4.17](#) and [Figure 4.18](#), the tool does indeed show a promising level of compatibility with TBB - it successfully identifies all four threads (TBB would have spun up three threads on top of the starting thread to work on the other three virtual cores), identifies a critical synchronisation stack trace on the lock, and also correctly summarises the lock.

The additional lock detected is likely an internal TBB synchronisation mechanism, which would explain the high criticality. Future investigation may be warranted to enhance the tool to understand this and summarise it.

4.5 Real-world Benchmarks & Programs

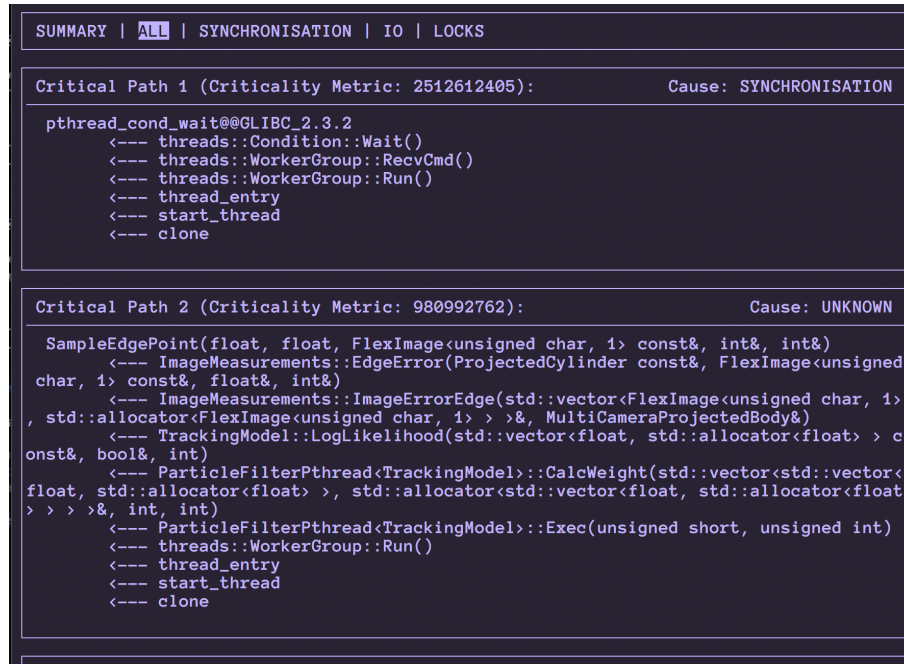
Moving on from contrived examples, in this section we attempt to evaluate the tool against well-known benchmarks and programs.

4.5.1 Parsec - Bodytrack

The Parsec benchmark [9] has a number of different tools, one of which is Bodytrack, which tracks a human body with a number of cameras across a sequence of images. From this part of the evaluation onwards

We ran the extended GAPP tool against it, using the Parsec command `bin/parsecmgmt -a run -p bodytrack -i simlarge` to execute Bodytrack with a large input size. It took around eight seconds to complete, and most of that time was processing individual frames.

Bodytrack is organised around a single thread sends commands to worker threads, which then process the commands and return to waiting. As a result, we believe the primary parallelism bottleneck in the program is going to be the worker threads waiting to be notified of new work to do.



```
SUMMARY | ALL | SYNCHRONISATION | IO | LOCKS

Critical Path 1 (Criticality Metric: 2512612405):          Cause: SYNCHRONISATION

pthread_cond_wait@GLIBC_2.3.2
<--- threads::Condition::Wait()
<--- threads::WorkerGroup::RecvCmd()
<--- threads::WorkerGroup::Run()
<--- thread_entry
<--- start_thread
<--- clone

Critical Path 2 (Criticality Metric: 980992762):          Cause: UNKNOWN

SampleEdgePoint(float, float, FlexImage<unsigned char, 1> const&, int&, int&)
<--- ImageMeasurements::EdgeError(ProjectedCylinder const&, FlexImage<unsigned
char, 1> const&, float&, int&)
<--- ImageMeasurements::ImageErrorEdge(std::vector<FlexImage<unsigned char, 1>
, std::allocator<FlexImage<unsigned char, 1> > >&, MultiCameraProjectedBody&)
<--- TrackingModel::LogLikelihood(std::vector<float, std::allocator<float> > c
onst&, bool&, int)
<--- ParticleFilterPthread<TrackingModel>::CalcWeight(std::vector<std::vector<
float, std::allocator<float> >, std::allocator<std::vector<float, std::allocator<float
> > >&, int, int)
<--- ParticleFilterPthread<TrackingModel>::Exec(unsigned short, unsigned int)
<--- threads::WorkerGroup::Run()
<--- thread_entry
<--- start_thread
<--- clone
```

Figure 4.19: The ‘all’ page for bodytrack.

SUMMARY ALL <u>SYNCHRONISATION</u> IO LOCKS	
Critical Path 1 (Criticality Metric: 2512612405):	
Cause: SYNCHRONISATION	
pthread_cond_wait@GLIBC_2.3.2 <--- threads::Condition::Wait() <--- threads::WorkerGroup::RecvCmd() <--- threads::WorkerGroup::Run() <--- thread_entry <--- start_thread <--- clone	
Waker (25%): __pthread_cond_broadcast <--- threads::Condition::NotifyAll() <--- threads::WorkerGroup::SendInternalCmd(int) <--- threads::WorkerGroup::SendCmd(unsigned short) <--- TrackingModelPthread::GaussianBlurPthread(FlexImage<unsigned char, 1>*, FlexImage<unsigned char, 1>*) <--- TrackingModelPthread::CreateEdgeMap(FlexImage<unsigned char, 1>*, FlexImage<unsigned char, 1>*) <--- TrackingModelPthread::GetObservation(float) <--- ParticleFilter<TrackingModel>::Update(float) <--- mainPthreads(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, int, int, int, int, int, bool) <--- main	
Waker (25%): __pthread_cond_broadcast <--- threads::Condition::NotifyAll() <--- threads::WorkerGroup::SendInternalCmd(int) <--- threads::WorkerGroup::SendCmd(unsigned short) <--- ParticleFilterPthread<TrackingModel>::CalcWeights(std::vector<std::vector<float, std::allocator<float>>, std::allocator<std::vector<float, std::allocator<float>>>> <--- ParticleFilter<TrackingModel>::Update(float) <--- mainPthreads(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, int, int, int, int, int, bool) <--- main <--- __libc_start_main <--- _start	
Waker (24%): __pthread_cond_broadcast <--- threads::Condition::NotifyAll() <--- threads::WorkerGroup::SendInternalCmd(int) <--- threads::WorkerGroup::SendCmd(unsigned short) <--- ParticleFilterPthread<TrackingModel>::GenerateNewParticles(int) <--- ParticleFilter<TrackingModel>::Update(float) <--- mainPthreads(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, int, int, int, int, int, bool) <--- main <--- __libc_start_main <--- _start	
Waker (12%): __pthread_cond_broadcast <--- threads::Condition::NotifyAll() <--- threads::WorkerGroup::SendInternalCmd(int) <--- threads::WorkerGroup::SendCmd(unsigned short) <--- TrackingModelPthread::CreateEdgeMap(FlexImage<unsigned char, 1>*, FlexImage<unsigned char, 1>*) <--- TrackingModelPthread::GetObservation(float) <--- ParticleFilter<TrackingModel>::Update(float) <--- mainPthreads(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, int, int, int, int, int, bool) <--- main <--- __libc_start_main	
Waker (5%): __pthread_cond_broadcast <--- threads::Condition::NotifyAll() <--- threads::WorkerGroup::SendInternalCmd(int) <--- threads::WorkerGroup::SendCmd(unsigned short) <--- TrackingModelPthread::GaussianBlurPthread(FlexImage<unsigned char, 1>*, FlexImage<unsigned char, 1>*) <--- TrackingModelPthread::CreateEdgeMap(FlexImage<unsigned char, 1>*, FlexImage<unsigned char, 1>*) <--- TrackingModelPthread::GetObservation(float) <--- mainPthreads(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>, int, int, int, int, int, bool) <--- main	

Figure 4.20: The ‘synchronisation’ page for bodytrack, showing wakers for the primary discovered bottleneck.

```

SUMMARY | ALL | SYNCHRONISATION | IO | LOCKS

Critical Path 1 (Criticality Metric: 21321824): Cause: IO

write
  <--- _IO_file_write@@GLIBC_2.2.5
  <--- new_do_write
  <--- __GI__IO_do_write
  <--- _IO_new_file_xsputn
  <--- fwrite
  <--- bool FlexSaveBMP8u<3>(char const*, FlexImage<unsigned char, 3>&)
  <--- bool FlexSaveBMP<unsigned char, 3>(char const*, FlexImage<unsigned char,
3>&)
  <--- TrackingModel::OutputBMP(std::vector<float, std::allocator<float> > const
&, int)
  <--- mainPthreads(std::__cxx11::basic_string<char, std::char_traits<char>, std
::allocator<char> >, int, int, int, int, int, bool)

Top files (with criticalities):
sequenceB_4/Result0003.bmp: 21321824

```

Figure 4.21: The ‘io’ page for `bodytrack`, showing wakers for the primary discovered bottleneck.

Our results, shown in [Figure 4.19](#), [Figure 4.20](#), and [Figure 4.21](#) are positive for our tool. The most critical stack trace reported is a synchronisation stack trace, and has been correctly classified.

As we see in the unabridged synchronisation output ([Figure 4.20](#)), there are many wakers of the single internal synchronisation stack trace, demonstrating the value of this feature - without this extra detail, there would be no indication of which parts of the application are doing synchronisation, as the primary critical stack trace is not hugely helpful. It also verifies our assumption about the location of the primary bottleneck.

There is also an IO stack trace, which was reported about one of the results files, which is helpful as it indicates that, if possible, it would be beneficial to asynchronously write results, rather than batching them all together.

The other stack traces are, as hinted at by two of the visible ones in [Figure 4.19](#), in parts of the code which do computation, like vector projection and multiplication. That some stack traces are not directly themselves part of the problem causing threads to be non-runnable is due to the fact that they are the positions most likely to be executing when other threads are blocking for the synchronisation. When other threads are blocked waiting, and the number of threads runnable is below the critical threshold, any time a thread’s time-slice naturally ends and it gets switched (but remains runnable) a critical stack trace is sent by GAPP. This can be a good thing - it helps identify *on-CPU* bottlenecks - but this relies on other threads running into real parallelism bottlenecks, and this sort of on-CPU analysis is best done with a specialised tool for that purpose.

We have also investigated how much overhead is added to this tool in [Figure 4.22](#). We see that a reasonable level of overhead - 6% - is added by using our tool on `bodytrace` linked against the system `glibc`. However, using our debug build of `glibc` in order to

Run type	Runs (s)	Average (s)	Overhead
Normal <code>glibc</code>	7.279, 7.291, 6.990	7.187	–
Normal <code>glibc</code> , with MGAPP	7.578, 7.657, 7.691	7.642	1.06x
Modified <code>glibc</code>	7.695, 7.412, 7.566	7.558	1.05x
Modified <code>glibc</code> , with MGAPP	7.878, 7.834, 7.810	7.841	1.09x

Figure 4.22: Comparison of `bodytrack` execution time with both the system `glibc` and our version built with debug symbols and frame pointer, as well as with and without the enhanced version of GAPP (MGAPP, (Modified GAPP) for brevity) tracing it.

be able to get full stack traces from library functions itself applies a notable overhead, and when we use our tool in addition to linking against the debug library we get a more significant overhead of nearly 10%. This is quite a lot, but for the level of utility you get from the tool we believe it is reasonable. The closest similar tool we are aware of, `wPerf`, has a typical overhead of around 5% [4]. It seems that this overhead is calculated against a program already built with debug symbols and frame pointers enabled, however, which would make our tool competitive with them - ours has a 4% overhead on this example when you compare the versions running a debug `glibc`.

4.5.2 Cuberite - A Minecraft Server

A real-world piece of software we looked at is Cuberite. Cuberite is a fan-made C++ implementation of the Minecraft [10] (a successful computer game) server protocol, and is a drop-in replacement for the standard Java server which Microsoft distribute with Minecraft. It is reasonably popular, with 2670 stars on GitHub [32], 140 contributors, and 534 issues, which indicates an active community.

Minecraft has a voxel-based world - that is to say, the game world is made up of a large grid, and a type of block exists in each cube of the grid. The world is randomly generated and essentially infinite, with an overground area and an underground area which has caves generated within it. The game has a number of dynamic features, including animals that can interact with the world (such as a sheep, pictured in Figure 4.23 having eaten some grass), a system known as ‘redstone’ which allows players to create dynamic circuits inspired by electronics, and a dynamic lighting system.

The lighting system is notorious among players of the game for causing slowdowns, as whenever a block is updated (such as being broken by a player or eaten by a sheep), the lighting is updated for that block. If it changes, any adjacent blocks are additionally updated. This can cause problems when as a single update can cause an explosion of lighting recalculations. A major cause of this is what is known as ‘lighting bugs’, which occur when a world is generated, where an underground cave is unlit despite having a light-emitting block such as lava contained within it. These issues can result in long delays, because a single lighting update can propagate across systems of underground

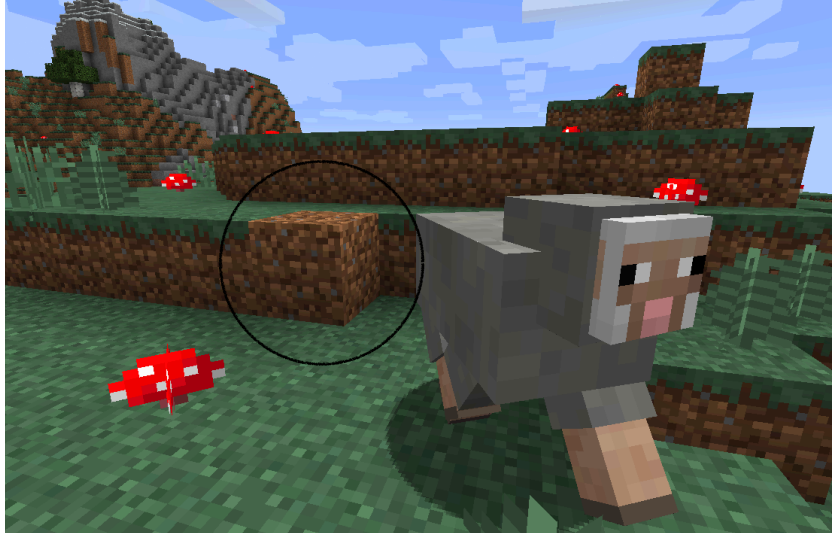


Figure 4.23: The Minecraft game world. The sheep that is visible in the picture ‘ate’ a square of grass, turning it into dirt (circled in black).

caves, causing many thousands of updates.

The ‘redstone’ system in the game can also cause similar issues, as if a wire is turned on or off, it all of the related electronic blocks nearby are updated to propagate the signal, which can also cause explosions of updates.

The Minecraft world is split into vertical slices named *chunks*, which are sixteen blocks wide and long and 256 blocks tall (there is no vertical slicing; the game world does not continue below zero or above 256 blocks). These ‘chunks’ are important as they are the granularity with which the world is communicated between the client and the server - the game clients request the chunks of the map nearby the player, and the server delivers them. In addition, they are also the granularity that the map is saved with.

Unlike the standard server distribution, the Cuberite server is multi-threaded, and therefore appropriate for testing with our tool. Cuberite synchronises the game world across threads using locks, which are granular at the level of the in-game chunks described above. This means that if two game update cause changes to the same chunk, one must wait for the other to finish.

We set up our tool to trace the server, and then loaded the server, connected as a player, and walked around for a few seconds before disconnecting and closing the server. One of the challenges of Cuberite compared to the rest of our evaluation above is that it is a real-world, large piece of software: nearly 150 thousand lines of C++ split across nearly 300 files [32], with a game world running dozens of threads updating dozens of times a second.

Our results are shown in [Figure 4.24](#) and [Figure 4.25](#), and are promising. Predominantly

SUMMARY ALL SYNCHRONISATION IO LOCKS	
Critical Path 1 (Criticality Metric: 2141000964):	Cause: SYNCHRONISATION
<pre>__lll_lock_wait <--- __gthread_recursive_mutex_lock(pthread_mutex_t*) <--- std::recursive_mutex::lock() <--- cCriticalSection::Lock() <--- cCSLock::Lock() <--- cCSLock::cCSLock(cCriticalSection&) <--- cChunkMap::GetBlockTypeMeta(int, int, int, unsigned char&, unsigned char&) <--- cWorld::GetBlockTypeMeta(int, int, int, unsigned char&, unsigned char&) <--- cIncrementalRedstoneSimulator::Simulate(float) <--- cSimulatorManager::Simulate(float)</pre>	
Critical Path 2 (Criticality Metric: 1309445993):	Cause: SYNCHRONISATION
<pre>__lll_lock_wait <--- __gthread_recursive_mutex_lock(pthread_mutex_t*) <--- std::recursive_mutex::lock() <--- cCriticalSection::Lock() <--- cCSLock::Lock() <--- cCSLock::cCSLock(cCriticalSection&) <--- cChunkMap::ChunkLighted(int, int, unsigned char const (&) [32768], unsign ed char const (&) [32768]) <--- cWorld::ChunkLighted(int, int, unsigned char const (&) [32768], unsigned char const (&) [32768]) <--- cLightingThread::LightChunk(cLightingThread::cLightingChunkStay&) <--- cLightingThread::Execute()</pre>	
Critical Path 3 (Criticality Metric: 1046576739):	Cause: SYNCHRONISATION
<pre>__lll_lock_wait <--- __gthread_recursive_mutex_lock(pthread_mutex_t*) <--- std::recursive_mutex::lock() <--- cCriticalSection::Lock() <--- cCSLock::Lock() <--- cCSLock::cCSLock(cCriticalSection&) <--- cChunkMap::AddChunkClient(int, int, cClientHandle*) <--- cWorld::AddChunkClient(int, int, cClientHandle*) <--- cClientHandle::StreamChunk(int, int, cChunkSender::eChunkPriority)</pre>	

Figure 4.24: The ‘all’ page of our tool run against Cuberite.

of note is that the tool works, and manages to identify a huge number of stack traces - in fact, it exposed a problem in our interface: some information is not pre-calculated, which makes scrolling slow, as it is recalculating it on every refresh as it is rendered. This issue was imperceptible for all of our contrived examples, and even Bodytrack, as they have so few stack traces these issues never arise.

Looking at the specifics, we see that the top reported critical stack traces are all synchronisation-related, and we see that one is within the lighting system. The most critical stack trace, however, has a criticality nearly double the next most critical, and is within the redstone (in-game electronics) system.

The unabridged synchronisation stack trace view is quite enlightening: 77% of the time, the waker of the redstone stack trace was the lighting system - an insight which would not have been possible without the extended stack trace information showing us who the wakers of this synchronisation bottleneck are.

This leads to a natural assumption that the real cause of this synchronisation bottleneck is the lighting system - not the redstone system as the abridged stack trace would imply.

Because of this stack trace, we investigated the source code, and found that there is a global lock which is used by the `ChunkMap` class. The lighting stack trace acquires this lock

The screenshot shows a tool's 'sync' page with a dark background and light text. At the top, there is a navigation bar with links: SUMMARY | ALL | SYNCHRONISATION | IO | LOCKS. Below this, a header indicates 'Critical Path 1 (Criticality Metric: 2141000964):' and 'Cause: SYNCHRONISATION'. The main content is divided into three sections: a critical path stack trace, a waker (77%) section, and a waker (10%) section. Each section lists a series of function calls in a stack trace format, with the most recent call at the top. The critical path section shows a lock wait followed by several calls to GetBlockTypeMeta. The waker (77%) section shows an unlock followed by calls to GetChunkData and ReadChunks. The waker (10%) section shows another unlock followed by calls to ChunkLighted and LightChunk.

```

SUMMARY | ALL | SYNCHRONISATION | IO | LOCKS

Critical Path 1 (Criticality Metric: 2141000964):          Cause: SYNCHRONISATION

__lll_lock_wait
<--- __gthread_recursive_mutex_lock(pthread_mutex_t*)
<--- std::recursive_mutex::lock()
<--- cCriticalSection::Lock()
<--- cCSLock::Lock()
<--- cCSLock::cCSLock(cCriticalSection&)
<--- cChunkMap::GetBlockTypeMeta(int, int, int, unsigned char&, unsigned char&)
)
<--- cWorld::GetBlockTypeMeta(int, int, int, unsigned char&, unsigned char&)
<--- cIncrementalRedstoneSimulator::Simulate(float)
<--- cSimulatorManager::Simulate(float)

Waker (77%):
__lll_unlock_wake
<--- __gthread_recursive_mutex_unlock(pthread_mutex_t*)
<--- std::recursive_mutex::unlock()
<--- cCriticalSection::Unlock()
<--- cCSLock::Unlock()
<--- cCSLock::~cCSLock()
<--- cChunkMap::GetChunkData(int, int, cChunkDataCallback&)
<--- cWorld::GetChunkData(int, int, cChunkDataCallback&)
<--- cLightingThread::ReadChunks(int, int)
<--- cLightingThread::LightChunk(cLightingThread::cLightingChunkStay&)

Waker (10%):
__lll_unlock_wake
<--- __gthread_recursive_mutex_unlock(pthread_mutex_t*)
<--- std::recursive_mutex::unlock()
<--- cCriticalSection::Unlock()
<--- cCSLock::Unlock()
<--- cCSLock::~cCSLock()
<--- cChunkMap::ChunkLighted(int, int, unsigned char const (&) [32768], unsigned
char const (&) [32768])
<--- cWorld::ChunkLighted(int, int, unsigned char const (&) [32768], unsigned
char const (&) [32768])
<--- cLightingThread::LightChunk(cLightingThread::cLightingChunkStay&)
<--- cLightingThread::Execute()

```

Figure 4.25: The ‘sync’ page of our tool run against Cuberite.

for a `GetChunkData`, which either loads or returns (if already loaded) a chunk. We believe that there is no reason that these methods could not use a more granular locking strategy to avoid this global lock contention which arises merely for loading individual chunks in. Notably, this lock is required for the primary stack trace we detected, which only uses the `GetBlockTypeMeta` method - a method which merely gets information about a specific block within a chunk, but requires a global lock to do so. This means that if the lighting thread is loading chunks to relight, other systems which only want a single block from an already-loaded chunk must wait until the lighting thread finishes loading - and vice-versa.

`ChunkMap` is a 2,000 line class, which uses this lock dozens of times, and so it is beyond the scope of this project to create a fix for this, but we strongly believe that a more granular approach to locking the chunk map would offer significant performance improvements. We tested a small patch which avoids the lighting system acquiring the lock (which would not be functionally correct, but would approximate the *maximum* performance gains from more granular locks), and reports the length of time it took to calculate lighting aggregated over several hundred calls (recorded in nanosecond precision). Our results are shown in [Figure 4.26](#), and show around a maximum of a 2x increase in performance for a lighting thread that does not need to deal with contention.

One problem that evaluating Cuberite exposes is it stretches the limitations of our

Time of updates	Locking	Number of updates	Avg (ms)	Improvement
Server initialisation	Y	200	8.22	—
Server initialisation	N	200	3.63	2.26x
Standard gameplay	Y	400	8.56	—
Standard gameplay	N	400	3.83	2.23x

Figure 4.26: Comparison of average time of chunk-based lighting updates with and without locking, to approximate an upper bound for improvements from increased lock granularity.

tool - with complex synchronisation activities, regular IO to load parts of the map, and a significant number of threads, it causes samples to be lost and error reporting which we built in to flag up errors indicating the back-end system got into a strange state (such as a `futex_wait` call taking place when the system thought one was ongoing). We attempt to quantify these issues in [section 4.6](#).

4.6 Quantifying errors

Our system relies on a number of imperfect mechanisms: in certain places, return probes are relied upon (this can be a problem, as explained in [subsection 3.3.1](#)), and in others the complicated state machine of the kernel’s behaviour is not fully reflected in our tracing. For example, we do not have any facility for tracking a futex which wakes due to a timeout. There are likely similar exceptions to our approach to correlating numerous events at the kernel level, and so we often end up in strange states, such as entering futex wait when we have state in our backend indicating a futex wait on that thread is *already* ongoing.

```
Error/warning from eBPF: Unexpected futex_wait (already waiting) (Error 2)
Error/warning from eBPF: Unexpected futex_wait (already waiting) (Error 2)
Error/warning from eBPF: Unexpected futex_wait (already waiting) (Error 2)
Error/warning from eBPF: Unexpected futex_wait (already waiting) (Error 2)
Error/warning from eBPF: Waker of a thread which was waiting on a futex was not recorded
(Error 4)
```

Figure 4.27: A few of the errors output by our evaluation of Cuberite.

Where these issues take place, we have a system in place which sends an error to the front-end through a perf buffer which is then printed. Our evaluation of Cuberite in particular caused a lot of these errors to be printed, and a snippet is shown in [Figure 4.27](#). We additionally clean up previous state and continue as normal in these calls, to avoid disruption.

4.6.1 Evaluating error count in Cuberite

To evaluate how significant these errors are, we re-run Cuberite with a modified version of our tracer which outputs the total number of futex-related errors, and the total number of futex-related traces reported. Since the futex traces are unconditional, we are then able to determine an exact ratio of how many times a futex call put our system into an unusual state. We found that while tracing Cuberite for around ten seconds, 3,848 futex events were reported, and 527 errors were reported - approximately 14%. This number is not insignificant, and warrants investigation to try to identify what cases, exactly, are leading to these errors, however it should not impact the integrity of the results significantly as the vast majority of events are still being processed correctly. Across the rest of our evaluation, we saw very few of these errors being reported, and so it seems reasonable to conclude that an upper bound for this type of error in the real world is around 15%.

4.6.2 Missing futex traces

For unknown reasons, the futex event table that is populated in the front-end sometimes has missing entries - for example, a futex wait stack trace is reported, but not the corresponding wake trace, or vice versa. This is handled in the front-end by ignoring any pairs without both stack traces populated, but is still important to be aware of and quantify.

We suspect some of the reasons this is true may be out of our hands - for example, a stack trace which cannot be translated for some reason - but that there may be cases where, for some reason, a wait or wake stack trace is not sent. This, again, may be due to our tracing not fully lining up to the full state machine of actions that code using the futex system call is capable of. For example, a thread waking on a futex would never be woken if the entire process is terminated, and so it would make sense to miss the corresponding waking event.

We modified our tool to report the number of futex events with at least one blank stack trace, and ran *bodytrack* with a large simulation. It found 156 total futex events, and 5 with a missing trace - around 3%. We also tried it with *multi_lock*, and found 1322 futex events, with 50 missing a trace - 3.7%.

Generally, these results are not hugely concerning to us, given there are a number of reasonable explanations for why certain events would be lost, the proportion of lost events is very low, and that our tool is correctly identifying, classifying, and detailing expected bottlenecks independently of the complexity of the applications we trial it on.

5 | Conclusion

5.1 Summary

Through this project, we have demonstrated that tracing at the kernel level can provide an effective substitute for traditional profiling mechanisms which operate at a higher level, introducing dependencies such as on a specific threading library. Even without the abstraction level of user-space libraries or instrumentation, we can recover useful information such as locking positions and file names and associate them with bottlenecks.

We have presented a tool which takes the techniques we have learnt and developed to avoid the pitfalls of tracing at such a low level, such as the unreliability of return probes ([subsection 3.3.1](#)), and presents the accumulated information in an accessible and straightforward user interface.

We have shown that the overhead of this tracing is sufficiently low to be competitive with the most similar profiling tools available, such as wPerf [4], which operates at a similarly low-level, and that it works even on complicated projects consisting of over a hundred thousand lines of real-world code ([section 4.5](#)), and across threading libraries - even ones heavily abstracted from basic threading, such as Intel's TBB.

5.2 Future Work

One of the the most exciting things about this project is the future work it helps lay the groundwork for - we have only scraped the surface of what is possible through kernel-level tracing, and far more information can be put together. A lot of the main difficulties of this project were discovering through trial and errors issues such as the return probe problems, and now that these are solved development of future work can be much easier. Some particularly prominent ideas we have for practical extensions are explored below.

5.2.1 DNS Probing

Trace DNS queries to associate future networking traces with a DNS name, rather than an IP address. This is particularly important in a world of predominantly cloud-hosted websites, where an IP address does not necessarily map to a specific website - while the IP addresses we provide with stack traces provide utility, this would be the next logical step to improving these.

5.2.2 Conflation of identically-defined locks

Extend our front-end lock processing logic to understand unique instances of the same lock - for example, two instances of a `List` class which utilises a coarse locking scheme would each have their own lock in memory, and our system would consider them two separate locks. While it is useful in some instances to treat them separately, some users may have thousands of lists throughout an application and prefer an aggregated view of all list locks.

We believe this would be non-trivial to achieve, but an attempt could be made through analysis of those locks whose waiting and waking stack traces are subsets of each-other. It may also be possible to employ static code analysis for this goal - while that would limit the languages our tool performed best without, other languages could continue to benefit from the other features we presented. It would also be interesting to investigate whether we can learn any lessons from the approach of wPerf to assist with this post-processing, as we should have a wider base of data to use than the authors had.

5.2.3 Ad-hoc synchronisation

Detecting and appropriately handling spin locks and other ad-hoc user-level synchronisation. In theory, sanely implemented spin locks will spin for a very brief period before synchronising with real `futex` operations, but users may make a false assumption that their locks aren't spinning for long. This could be done through automated instrumentation and code analysis - we have shown [subsection 2.5.2](#) that only a small subset of assembly can correctly implement synchronisation, so a loop-back pattern involving these instructions could be classified as a spinlock.

5.2.4 Unit testing

A unit testing suite built around BCC. The biggest difficulty - by far - in developing this project was dealing with bugs and issues that arose and were difficult to debug because of the low-level we were dealing with.

This could be implemented by, for example, having a user-level virtual machine simulating the eBPF virtual machine bytecode generated, and having a high-level interface for defining what probes should be triggered and what output should be expected in the eBPF maps after the probes are processed.

We believe that a tool like this would be a significant undertaking, but be a huge utility for future development of this tool and others like it, as well as an asset to the wider eBPF community.

Bibliography

- [1] Intel. Intel® core™ i9-9900kf processor specification, 2019. [Accessed 10 June]. URL: <https://ark.intel.com/content/www/us/en/ark/products/190887/intel-core-i9-9900kf-processor.html>.
- [2] Intel. Intel® core™ i7-990x processor extreme edition specification, 2011. [Accessed 10 June]. URL: <https://ark.intel.com/content/www/us/en/ark/products/52585/intel-core-i7-990x-processor-extreme-edition.html>.
- [3] Paul Alcorn. Cpu hierarchy 2019: Intel and amd processors ranked, 2019. [Accessed 16 January]. URL: <https://www.tomshardware.com/reviews/cpu-hierarchy,4312.html>.
- [4] Fang Zhou, Yifan Gan, Sixiang Ma, and Yang Wang. wperf: Generic off-cpu analysis to identify bottleneck waiting events. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 527–543, Carlsbad, CA, 2018. USENIX Association. URL: <https://www.usenix.org/conference/osdi18/presentation/zhou>.
- [5] Adarsh Yoga and Santosh Nagarakatte. A fast causal profiler for task parallel programs. *CoRR*, abs/1705.01522, 2017. URL: <http://arxiv.org/abs/1705.01522>, [arXiv:1705.01522](https://arxiv.org/abs/1705.01522).
- [6] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield. Analyzing lock contention in multithreaded applications. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '10, pages 269–280, New York, NY, USA, 2010. ACM. URL: <http://doi.acm.org/10.1145/1693453.1693489>, [doi:10.1145/1693453.1693489](https://doi.org/10.1145/1693453.1693489).
- [7] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. *CoRR*, abs/1608.03676, 2016. URL: <http://arxiv.org/abs/1608.03676>, [arXiv:1608.03676](https://arxiv.org/abs/1608.03676).
- [8] Reena Hiran. PhD thesis, Imperial College London, (to appear).
- [9] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

- [10] Mojang. Minecraft. [Accessed 15 June]. URL: <https://www.minecraft.net/>.
- [11] Murray Woodside, Greg Franks, and Dorina C. Petriu. The future of software performance engineering. In *2007 Future of Software Engineering*, FOSE '07, pages 171–187, Washington, DC, USA, 2007. IEEE Computer Society. URL: <https://doi.org/10.1109/FOSE.2007.32>, doi:10.1109/FOSE.2007.32.
- [12] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muza-hid. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 298–313, New York, NY, USA, 2017. ACM. URL: <http://doi.acm.org/10.1145/3064176.3064186>, doi:10.1145/3064176.3064186.
- [13] Dan Luu. Sampling v. tracing. [Accessed 25 January]. URL: <https://danluu.com/perf-tracing/>.
- [14] Brendan Gregg. Off-cpu analysis. [Accessed 25 January]. URL: <http://www.brendangregg.com/offcpuanalysis.html>.
- [15] Robert J. Hall. Call path profiling. In *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 296–306, New York, NY, USA, 1992. ACM. URL: <http://doi.acm.org/10.1145/143062.143147>, doi:10.1145/143062.143147.
- [16] Nathan R. Tallent and John M. Mellor-Crummey. Effective performance measurement and analysis of multithreaded applications. *SIGPLAN Not.*, 44(4):229–240, February 2009. URL: <http://doi.acm.org/10.1145/1594835.1504210>, doi:10.1145/1594835.1504210.
- [17] G. Contreras and M. Martonosi. Characterizing and improving the performance of intel threading building blocks. In *2008 IEEE International Symposium on Workload Characterization*, pages 57–66, Sep. 2008. doi:10.1109/IISWC.2008.4636091.
- [18] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual - Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4*, November 2018.
- [19] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, January 1990. URL: <https://doi.org/10.1109/71.80120>, doi:10.1109/71.80120.
- [20] Ulrich Drepper. Futexes are tricky, 07 2004.
- [21] Brendan Gregg. Linux extended bpf (ebpf) tracing tools. [Accessed 23 January]. URL: <http://www.brendangregg.com/ebpf.html>.
- [22] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference*

- Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association. URL: <http://dl.acm.org/citation.cfm?id=1267303.1267305>.
- [23] Brendan Gregg. Bpf: Tracing and more. Linux Conf Au 2017 - Hobart, Australia, 2017. URL: <https://www.youtube.com/watch?v=JRFNIKUROPE>.
 - [24] Jonathan Corbet. Uprobes in 3.5, 2012. [Accessed 16 January]. URL: <https://lwn.net/Articles/499190/>.
 - [25] Mathieu Xhonneux, Fabien Duchene, and Olivier Bonaventure. Leveraging ebpf for programmable network functions with ipv6 segment routing. *CoRR*, abs/1810.10247, 2018. URL: <http://arxiv.org/abs/1810.10247>, [arXiv:1810.10247](https://arxiv.org/abs/1810.10247).
 - [26] Mathieu Xhonneux and Olivier Bonaventure. Flexible failure detection and fast reroute using ebpf and srv6. *CoRR*, abs/1810.10260, 2018. URL: <http://arxiv.org/abs/1810.10260>, [arXiv:1810.10260](https://arxiv.org/abs/1810.10260).
 - [27] Panchamukhi Keniston and Hiramatsu. Kernel probes (kprobes). [Accessed 29 May]. URL: <https://www.kernel.org/doc/Documentation/kprobes.txt>.
 - [28] Ashley J Davies-Lyons and Paul Chaignon. kretprobe not consistently being triggered for sys_futex calls?, Feb 2019. URL: <https://lists.iovisor.org/g/iovvisor-dev/topic/29702757#1570>.
 - [29] Linus Torvalds. Linux kernel (v5.1.5). <https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?h=v5.1.5>, 2019.
 - [30] IOVisor. Bcc, 2019. URL: <https://github.com/iovisor/bcc>.
 - [31] Inc Netflix. Execsnoop. <https://github.com/iovisor/bcc/blob/master/tools/execsnoop.py>, 2016.
 - [32] cuberite. Cuberite, 2019. URL: <https://github.com/cuberite/cuberite>.

A | Additional Screenshots

A full set of screenshots of the final tool corresponding to the same run as the screenshots presented in [subsection 3.1.4](#) follow. Some of the screenshots correspond to different runs of the tool to assist in demonstrating all of the features within a relatively compact number of screenshots.

```
GAPP Results | bin/multi_mutex | 'q' to quit

SUMMARY | ALL | SYNCHRONISATION | IO | LOCKS

73 threads were discovered and tracked.

Criticality Metric per thread:
  17284: 2152979511
  17195: 1925093022
  17261: 1735524246
  17302: 1667026873
  17262: 1639589845
  17193: 1312448277
  17304: 1299973384
  17299: 1299814198
... and 65 more.
Total: 13032449356

Total switches: 1480
Critical switches: 1404
Post Processing time (ms): 173

Top critical functions and lines, with frequency:

  big_work(int) -- 6195
    multi_mutex.cpp:16 -- 4879
    multi_mutex.cpp:15 -- 1311
    multi_mutex.cpp:18 -- 5

  medium_work(int) -- 29
    multi_mutex.cpp:27 -- 21
    multi_mutex.cpp:26 -- 8

  small_work(int) -- 7
    multi_mutex.cpp:38 -- 6
    multi_mutex.cpp:37 -- 1
```

Figure A.1: The start of the summary screen, showing some summary statistics which were largely present in the original GAPP version; the only differences are the user interface around them and only listing the eight most critical threads.

```
GAPP Results | bin/multi_mutex | 'q' to quit

SUMMARY | ALL | SYNCHRONISATION | IO | LOCKS

multi_mutex.cpp:26 -- 8
small_work(int) -- 7
multi_mutex.cpp:38 -- 6
multi_mutex.cpp:37 -- 1

Most critical futex wakes:

Criticality 676141135:
  big_work(int) at multi_mutex.cpp:16
    <--- random_work(int) at multi_mutex.cpp:46
    <--- << C++ Thread Initialisation >>

Criticality 501889624:
  big_work(int) at multi_mutex.cpp:16
    <--- random_work(int) at multi_mutex.cpp:46
    <--- << C++ Thread Initialisation >>

Criticality 403068168:
  big_work(int) at multi_mutex.cpp:16
    <--- random_work(int) at multi_mutex.cpp:46
    <--- << C++ Thread Initialisation >>

Criticality 377839218:
  __lll_unlock_wake
    <--- std::mutex::unlock() at std_mutex.h:122
    <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
    <--- big_work(int) at multi_mutex.cpp:21
    <--- random_work(int) at multi_mutex.cpp:46
    <--- << C++ Thread Initialisation >>

Criticality 333840434:
  __lll_unlock_wake
    <--- std::mutex::unlock() at std_mutex.h:122
    <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168
    <--- big_work(int) at multi_mutex.cpp:21
    <--- random_work(int) at multi_mutex.cpp:46
    <--- << C++ Thread Initialisation >>
```

Figure A.2: The rest of the summary screen, having scrolled down, showing the most critical futex wake actions, & the most critical file activities.

GAPP Results bin/mutex_and_writes 'q' to quit	
SUMMARY ALL SYNCHRONISATION IO LOCKS	
Critical Path 1 (Criticality Metric: 34921084):	Cause: IO
__GI___writev	
Critical Path 2 (Criticality Metric: 28532781):	Cause: IO
__GI___close_nocancel	
Critical Path 3 (Criticality Metric: 20733959):	Cause: SYNCHRONISATION
__lll_lock_wait <--- std::mutex::lock() at std_mutex.h:103 <--- std::lock_guard<std::mutex>::lock_guard(std::mutex&) at std_mutex.h:162 <--- mutex_work(int) at mutex_and_writes.cpp:14 <--- << C++ Thread Initialisation >>	
Critical Path 4 (Criticality Metric: 10475115):	Cause: UNKNOWN
__GI___IO_un_link	
Critical Path 5 (Criticality Metric: 4368574):	Cause: UNKNOWN
_int_malloc	

Figure A.3: The top of the ‘all’ page, which indiscriminately shows any detected stack traces, but omits extra details like file names and additional associated synchronisation stack traces

GAPP Results bin/mutex_and_writes 'q' to quit	
SUMMARY ALL SYNCHRONISATION IO LOCKS	
Critical Path 1 (Criticality Metric: 20733959):	Cause: SYNCHRONISATION
__lll_lock_wait <--- std::mutex::lock() at std_mutex.h:103 <--- std::lock_guard<std::mutex>::lock_guard(std::mutex&) at std_mutex.h:162 <--- mutex_work(int) at mutex_and_writes.cpp:14 <--- << C++ Thread Initialisation >>	
Waker (100%): __lll_unlock_wake <--- std::mutex::unlock() at std_mutex.h:122 <--- std::lock_guard<std::mutex>::~~lock_guard() at std_mutex.h:168 <--- mutex_work(int) at mutex_and_writes.cpp:15 <--- << C++ Thread Initialisation >>	

Figure A.4: The synchronisation stack traces page, showing the associated ‘waker’ stack trace which woke the displayed stack trace.

GAPP Results bin/mutex_and_writes 'q' to quit		
SUMMARY ALL SYNCHRONISATION IO LOCKS		
Critical Path 1 (Criticality Metric: 34921084):		Cause: IO
__GI___writev		
Top files (with criticalities):		
out/10.txt: 9214496	out/7.txt: 5745727	out/9.txt: 5668161
out/8.txt: 5218309	out/6.txt: 3274669	out/11.txt: 3208233
out/3.txt: 100422		
Critical Path 2 (Criticality Metric: 28532781):		Cause: IO
__GI___close_nocancel		
Top files (with criticalities):		
out/11.txt: 17292240	out/8.txt: 4519006	out/10.txt: 3742872
out/7.txt: 2823905		

Figure A.5: The IO stack traces page, showing the associated files that were detected from the stack traces.

B | Evaluation Code Listings

This appendix contains additional code listings for the evaluation where they are omitted due to similarity to a different code listing.

```
1 static std::mutex work_mutex, work_mutex_two;
2
3 void __attribute__((noinline)) big_work(int id) {
4     std::lock_guard<std::mutex> l(work_mutex_two);
5     volatile int x = 5;
6     for (int i = 0; i < id * 10000000; i++) {
7         // calculations...
8     }
9 }
10 void medium_work(int id); // as big_work, but * 25000 in loop
11 void small_work(int id);  // as big_work, but * 5000 in loop
12
13 void random_work(int id) {
14     for (int i = 0; i < 10; i++) {
15         int r = rand() % 100;
16         if (r < 33) small_work(id);
17         else if (r < 66) medium_work(id);
18         else big_work(id);
19     }
20 }
21
22 int main() {
23     for (int t = 0; t < 8; t++) {
24         std::vector<std::thread> threads;
25         for (int i = 0; i < 18; i++) {
26             threads.emplace_back(random_work, i);
27         }
28         for (auto &thread : threads) thread.join();
29     }
30     return 0;
31 }
```

Listing B.1: multi_lock, an example program with multiple locks

```

1 void __attribute__((noinline)) big_work(int id) {
2     std::ofstream out_file("out/"
3         + std::to_string(id) + ".txt");
4     std::stringstream fileout;
5     for (int i = 0; i < id * 1000000; i++)
6         fileout << "Hello world";
7     out_file << fileout.str() << std::endl;
8     out_file.close();
9 }
10
11 int main() {
12     for (int t = 0; t < 4; t++) {
13         std::vector<std::thread> threads;
14         for (int i = 0; i < 8; i++)
15             threads.emplace_back(big_work, i);
16         for (auto &thread : threads) thread.join();
17     }
18     return 0;
19 }

```

Listing B.2: main_write, an example program for testing read IO detection