Distributed Virtual Time Execution of Programs

Author:
Oliver Myerscough

Supervisor:
Dr. Antony J Field

June 16, 2015

Abstract

Virtual time execution is a performance engineering technique which produces a speculative performance profile of a program under hypothetical optimisations without actually modifying the program's code. We present extensions to the existing Virtual Time Framework which explore a loosely synchronised approach to distributed virtual time executions. We show empirically that this approach can provide performance predictions within the accuracy of the underlying framework (approx. 8%), but is sensitive to errors introduced in other parts of the framework. The Virtual Time Framework uses a form of *parallel discrete-event simulation* where a each simulated core has a corresponding virtual timeline. By virtue of its loose coupling of virtual timelines between simulation nodes, our synchronisation scheme typically introduces an additional runtime overhead of less than 5% for simulations of small systems (fewer than 100 nodes) and tractable overheads of 50 - 75% for larger systems. Our implementation is reliable and is capable of simulating complex real world applications. We briefly use it to evaluate an application running in the *Jetty* Java servlet container.

Contents

Chapter 1

Introduction

Optimisations in complex applications may exploit non-trivial features of the program's execution profile, such as I/O contention and caching, which are difficult for even a well informed person to reason about. Changes to a program may cause it to run under a different schedule and, for example exhibit different locking or I/O patterns. These could lead to a significant changes in performance.

The Virtual Time Framework (VTF) is a performance analysis tool [6] which aims to inform developers about the potential effects of optimisations before they are implemented by running an execution-driven simulation of the modified program. VTF simulates faster code by allowing an accelerated function to run for more CPU time that others in order to progress the same amount of "virtual" time. A function accelerated by a factor of two is run for 200ms of real time to progress through 100ms of virtual time.

This poses a synchronisation problem in virtual time simulations of systems with more than one processing element. If the accelerated function above is run in a simulation of a dual core processor then the core running the accelerated code will advance through virtual time at half the rate of the other (as it takes it 200ms of real time to progress 100ms in virtual time). A synchronisation scheme is needed to keep simulation cores from diverging in virtual time. This simulation is a form of *discrete-event simulation* where cores (corresponding to virtual timelines) have threads scheduled on them (corresponding to events). The synchronisation problem in VTF is very similar to the *parallel discrete-event simulation* synchronisation problem for multiple virtual timelines.

During the course of his PhD, Nick Baltas developed two synchronisation schemes for VTF which allowed virtual time simulations on multicore platforms. The aim of this project is to develop a synchronisation scheme which would allow VTF to profile a distributed system in virtual time. With this in place, developers of distributed systems can explore hardware of software performance optimisations without having to modify existing code. In the context of a multi-tier web application, the Virtual Time Framework could be used to investigate the effect of deploying faster back end services on front end page loads in order to select which ones to optimise.

1.1 The Idea

The key challenge in this project has been to balance synchronisation of distributed simulation progress with simulation overhead. Our proposed solution is to run the simulations of each individual process in a form of lock step, so that the maximum possible (virtual) time difference between any two simulated processes is bounded. We accept the small errors this loose synchronisation scheme introduces and show that they can be made small enough to get results accurate to within the prediction error of VTF whilst maintaining a simulation overhead similar to that of the single process simulations.

1.2 Objectives

This project is an investigation into how the Virtual Time Framework (VTF) can be extended to support simulations of distributed systems. In it, we have aimed to investigate and understand in detail a particular approach to solving the synchronisation problem without having to resort to the expensive mechanisms usually used in parallel discrete-event simulation such as rollbacks.

1.3 Contributions

By adapting one of the multicore synchronisation schemes introduced in [6] we extend VTF to run loosely synchronised simulations of distributed systems without the need for expensive rollbacks or a "global" virtual timeline. The synchronisation scheme introduces a bounded maximum error, a parameter which is tuneable by the user.

We empirically assess the accuracy, simulation overhead and limitations of our synchronisation scheme. By experimenting with a number of "toy" examples, we find that it facilitates predictions which are accurate to within the VTF prediction error of 8%, but can exaggerate smaller errors introduced by other components of VTF. We find the additional simulation overhead introduced by our extensions to be negligible for small distributed systems of fewer than 100 nodes. The overhead remains reasonable (50 - 75%) for medium size systems (100 - 200 nodes).

We briefly use VTF to evaluate an application running in the *Jetty* servlet container, showing that our extensions are compatible a non-trivial program. In this case we achieve a low simulation overhead of 22%, within the range typically introduced by VTF on single node platforms, but limited prediction accuracy.

Chapter 2

Background

2.1 Simulation

Simulation means many different things to different people. In this project we mix forms of *parallel discrete-event simulation* and *execution driven simulation*.

2.2 Discrete-Event Simulation

Discrete-event simulation models a system as a sequence of events ordered in time. Events are mapped onto a timeline and occur instantaneously, one at a time and in increasing order of time. Events may alter system state when they occur and may schedule other events to occur after themselves.

DES systems use *virtual time* in the sense that the real time between two events occurring in the system bears no relationship with the simulations time at which those two events occur. The DES system will maintain a virtual time clock which tracks the virtual time of the most recently serviced event.

Discrete-event simulation algorithms can take either a *time stepped* or *event driven* form [1]. A time stepped algorithm iteratively advances simulation time by a constant time step and updates state variables to account for the step forward in time. Event driven algorithms maintain an ordered queue of events representing the timeline. On each iteration the algorithm takes the next event from the head of the queue, leaps time forward to the events occurrence time and executes the event. Using an event driven algorithm is advantageous for simulations which exhibit large quiet periods where no events are scheduled as these will simply be leapt over. In contrast, a time stepped algorithm would progress through quiet periods at the same rate in real time as it did through busy ones.

2.3 Parallel Discrete Event Simulation

A parallel discrete-event simulation (PDES) is composed of multiple logical processes (LPs), each with its own timeline onto which events are mapped and its own virtual time clock.

When an event occurs it may schedule subsequent events to occur on other LPs. Fujimoto [1] presents the analogy of a system of airports. Each airport is an LP. Planes taking off and landing are events. When a take off event occurs on one airport process, it schedules a landing event at the destination airport's process. The concept of multiple processes exchanging messages using virtual time as a synchronisation technique was formalised by Jefferson [2].

2.3.1 The PDES Synchronisation Problem

When executing a parallel discrete-event simulation, it is desirable to exploit concurrency between LPs. However, events at each LP must still occur in timeline order. A PDES system could serialise the events of each LPs timeline onto a single master timeline and execute them sequentially, but this would waste an obvious opportunity for parallelism. If each process serviced events in parallel with no synchronisation, at any given point in real time the virtual times of two LPs could be vastly different. This could lead to a situation where an event on LP A attempts to schedule another event on LP B at time which LP B already passed. This situation is known as a *causality error*.

What the PDES system does next will effect simulation accuracy. Continuing to run past a causality error would mean executing events out of order and may lead to incorrect simulation results as it may include a series of events which could not possibly happen. In order to be correct the simulator should use a checkpointing and rollback system to revert the changes made by processing events since the time at which the new event should arrive and re-run the simulation with the new event included. Rolling back an LP involves reverting its state variables to a checkpointed state and sending "anti-messages" to cancel the events it scheduled on other LPs timelines since the last checkpoint. If an LP receives an anti-message for an event it has already processed it must rollback to a checkpointed state from before that event was processed. In this way, one rollback can trigger another.

2.3.2 Synchronisation Algorithms

Because rollbacks are so expensive a PDES system must employ some synchronisation technique to reduce their frequency or avoid them entirely. Synchronisation algorithms for doing this can be broadly split into two categories.

Conservative algorithms avoid causality errors entirely, so never rollback. They will only allow a scheduled event to be run if there is no chance of a causality error occurring in the future as a result of taking that step. Most conservative synchronisation algorithms are based on the concept of "lookahead". The lookahead of a simulation is the smallest possible amount of time an event can schedule another event forward in time on another process's timeline. In the airports example, this would be the minimum flight time. The lookahead is used to define a safe interval in which events can be processed without any risk of causality errors. Let T_s be the minimum time of all processes in the simulation and L the lookahead. Therefore events in the interval $[T_s, T_s + L]$ are safe to process, as no new event can be scheduled in that interval.

Optimistic algorithms try to avoid introducing causality errors but may not do so

perfectly so must still detect and correct causality errors as they occur. An optimistic algorithm would typically employ some throttling heuristic to limit the progress of faster processors while still allowing parallelism.

2.4 Execution Driven Simulation

An execution driven simulation makes predictions by integrating a simulator into a program to model some behaviour. For example, Callgrind [3] runs an execution driven simulation of the CPU cache in order to predict hit rates. The program is run with instrumentation around all memory accesses. This instrumentation calls into a cache simulator, which is used to produce a hit rate prediction.

2.5 Network Emulation

Time dilation is presented in [4] as a technique for providing the illusion of a faster external world to a host OS running as a virtual machine. This is applied to network stack experimentation. Time dilation is implemented by modifying a hypervisor to scale the frequency of the timer interrupt passed to the host. For example, a virtual machine with a time dilation factor of 10 would have its timer interrupt frequency scaled down by 10 times, so a 1Gbps network interface will appear to process 10Gb of data per second.

A knock on effect of this clock scaling is that the host would experience 10 times as many CPU cycles per subjective second. Further work extends [5] the time dilation framework to allow independent scaling of CPU and disk resources. In this way, time dilation can be used to emulate arbitrary hardware at a cost of real time.

2.6 The Virtual Time Framework

The Virtual Time Framework (VTF) can be described as a *scheduling profiler*, in that it is a profiler which controls the schedule of its profiling target. This allows it to produce a speculative profile of a program were some parts of it to have different performance characteristics. The virtual time execution it performs is a form of execution driven simulation. The functional characteristics of the simulation are derived from executing the real code of the program under test, but performance characteristics (i.e. runtimes of individual methods) may be scaled by a linear factor or replaced with arbitrary performance models.

The majority of VTF was implemented by Nick Baltas over the course of his PhD [6] at Imperial College. This chapter gives an overview of the project and summarises the parts of his work which are most relevant to the distributed synchronisation scheme we have developed.

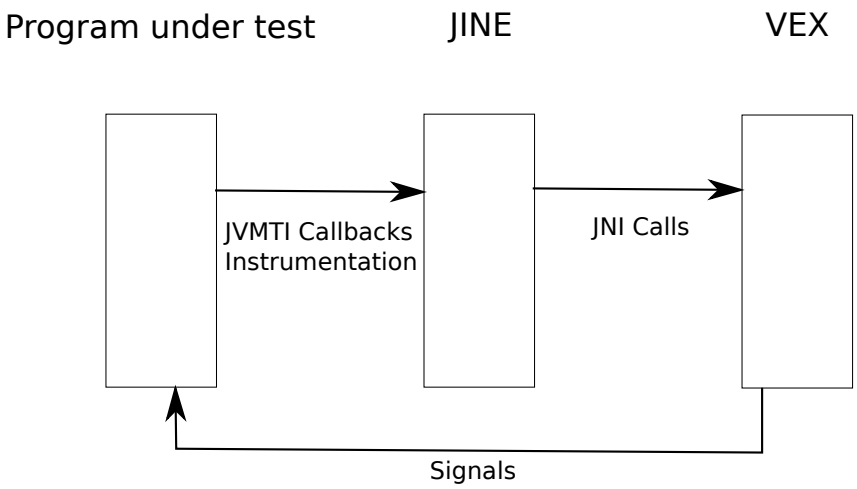

Figure 1: Information flow in VTF.

2.6.1 Project Structure

The core logic of VTF is implemented in C++ as a library, *VEX*, shown at the right in Figure 1. Code within this library is responsible for maintaining a model of the threads in the current simulation, controlling the simulation and producing a virtual time profile to be reported on completion. VEX exposes an API which a runtime specific instrumentation layer can use to pass in simulation state changes.

There are currently two instrumentation engines available: Java Instrumentation Engine (JINE) [7] and C/C++ Instrumentation Engine (CINE) [8]. During the course of this project we have focused solely on extending JINE to work with new network I/O callbacks in VEX.

2.6.2 JINE

As shown in Figure 1, JINE acts as an adapter between the VEX library and a program running on the JVM. JINE performs both static and dynamic instrumentation of Java classes by inserting additional bytecode instructions using the ASM [9] JVM bytecode engineering library. JINE also uses the Java Virtual Machine Tool Interface to trigger additional calls into VEX. These events call into VEX via the Java Native Interface (JNI). JINE is implemented in Java and C++.

Dynamic instrumentation is performed at runtime, at the classload stage, and is used to insert instruments which belong in all classes. Method entry and exit instruments, which allow VEX to measure method runtimes, are an example of such instrumentation. This is implemented using the Javaagent API, which allows a `ClassTransformer` to be registered to transform every loaded class.

However, system classes are loaded before the JINE Javaagent is initialised so cannot

be transformed in this way. Since these classes handle operations such as I/O which must be dealt with specially in virtual time execution they require instrumentation. JINE performs static instrumentation of specific system classes ahead of program execution. The instrumented classes are preloaded on simulation JVM startup so take priority over the uninstrumented versions.

2.6.3 VEX

The lowest layer of the VTF project is VEX, which implements a virtual time simulation engine in a general way, shown on the right in Figure 1. VEX is implemented in C++. The codebase consists of over 24,000 lines of code split over 91 files.

Key Implementation Elements

The `ThreadManager` classes hold much of the logic for controlling threads in virtual time and deciding which thread to schedule then. VEX will create one `ThreadManager` for each core in the simulation; each `ThreadManager` represents the "active" part of the simulated core. The `ThreadManager` is responsible for selecting an application thread to run, allowing it to run (scheduling it) and suspending it when the threads virtual timeslice has elapsed. Different synchronisation policies are implemented as `ThreadManager` subclasses.

The other half of a simulation core is its state, kept in a `VirtualTimeline` instance, which keeps a virtual timeline for each core in the simulation. Different implementations are used for single and multicore platforms (which will have different numbers of virtual timelines).

For each thread in the simulated application, VEX keeps a `ThreadState` object which holds information concerning the condition of the thread. Examples include whether it is running, suspended or waiting on a lock, the threads current virtual time and a selection of timers used by the profiler to measure time spent running application code.

Handling Events from JINE

VEX receives notification from the instrumented program in an event based fashion. These are used to update simulation state tracked by VEX. The instrumentation is placed at key points in the program, such as on method entry and exit. On method exit, the instrumentation will call into JINE, which will make a call into VEX via the JNI. There are two classes in VEX which might receive calls from JINE: `ThreadEventsBehaviour` and `MethodEventsBehaviour`. In the case of a method exit, since the event concerns a particular thread, a call is made to the `ThreadEventsBehaviour` class.

To handle this event, the `ThreadEventsBehaviour` finds the `ThreadManager` responsible for the thread which originated the event and passes control to it. The `ThreadManager` needs up take a measurement of the method duration in virtual time and pass it to the profiler module. This is done by inspecting the timers stored in the running thread's `ThreadState` instance. The elapsed time is also committed to the `VirtualTimeline`, which tracks the

virtual times of each core in the simulation.

2.6.4 Virtual Time Execution on a Single Core

In this section, we describe how VTF will run a simulation of a program running on a single core to make performance predictions.

The VTF Scheduler

VTF enforces a correct schedule in virtual time on its profiling target. This is a schedule that the program could be run under if the runtime of the code did conform to any time scalings or performance models that the simulation has specified.

Each thread in a virtual time execution will have a corresponding *time scaling factor* (TSF) which defines the rate at which virtual time passes with respect to the real execution time of that thread. A thread's TSF is dynamically calculated according to method time scaling specifications provided by the user. It will change as the thread enters and exits methods according to the time scalings that the user has applied to them. A TSF of more than 1 simulates code which is faster than reality, and less than 1 simulates slower code.

A single core simulation schedule threads execution events on a single timeline representing the CPU. The virtual time of the simulation vt_s is the virtual time at which the last thread was suspended. Runnable threads are kept in a priority queue ordered by virtual time. The next thread to schedule is simply the thread at the head of the queue. Supposed this thread has virtual time vt_i. This thread will be run starting at $max(vt_s, vt_i)$ for one timeslice in virtual time, which is $timeslice * TSF$ milliseconds of real time. After this real time has elapsed the thread will be suspended and one timeslice added to its virtual time.

The VTF scheduler uses a fair, priority free round-robin algorithm similar to the Borrowed Virtual Time algorithm [10]. We note that this algorithm is different to the Completely Fair Scheduler used by the Linux kernel since version 2.6.23 [11]. The effect this has on prediction accuracy is an open question, but appears to be small.

Implementation

The VTF scheduler lives entirely in userspace. It forces the OS scheduler's hand by ensuring that the only thread which is ever runnable is the one VTF has selected to run next. This is achieved by installing a signal handler in the application under test which allows VTF to block selected application threads. When the scheduler wishes to suspend a application thread, it will send a `SIGHUP` to that thread using the Linux `tkill` system call. The signal handler will block on a condition variable specific to this application thread, suspending it. This allows VTF to control which threads the OS will be able to schedule which lets it enforce a schedule in virtual time. In order to resume a blocked application thread the VTF scheduler will signal on that thread's condition variable, unblocking the thread so that it can once again be scheduled by the OS.

Any locks held by an application thread when it is suspended by VTF will remain under its possession until it is resumed and runs far enough to release them. From the application's point of view this is the correct behaviour, but it has the potential to cause deadlocks in the scheme described above. Consider a shared resource of both the VTF scheduler and an application thread (i.e. standard out). If the application thread is suspended by the scheduler while it holds a lock on the shared resource the scheduler system will deadlock if the scheduler attempts to acquire the lock on that resource before it reschedules the application thread. The scheduler will be waiting for a lock on a shared resource, which is held by an application thread. The thread is waiting for a signal on a condition variable which should be sent by the scheduler, completing the cycle.

To avoid this deadlock, each application thread has an additional "shared resource" lock associated with it. The application thread will only attempt to access shared resources while it holds this lock. Similarly, the scheduler will only attempt to suspend a thread if it holds that thread's shared resource lock. As a result, the scheduler will never suspend a thread which holds locks on resources shared with it so is safe from deadlocks.

An Example

Consider a simulation of two threads, a producer and consumer, running on a single core. The producer iteratively places values on a queue. The consumer polls values of this queue. Under VTF, the pair will be alternately scheduled for a single virtual timeslice. Figures 2 and 4 show *real* time scheduling traces of the two threads executed under a *virtual* time schedule with and without time scaling.

The size of real timeslice given to a thread is its *time scaling factor* multiplied by the virtual timeslice. In the absence of time scaling $TSF = 1$ for both threads so each will get equal size real timeslices as is shown in Figure 2. Over the course of this project we have developed a novel way of visualising program executions in both virtual and real time using two dimensional plots. These are a natural and information-rich way to represent events which occur on two timelines (real time and virtual time), and are used throughout this report to present complex situations clearly. Figure 3 shows the progress of the consumer and producer threads as a line through a series of coordinates in real time on the x axis and virtual time on the y axis. Observe that a horizontal line indicates progress in real time but not virtual time, i.e. a thread is not running. A vertical line indicates a thread leaps forward in virtual time at in instant in real time. This occurs when a thread is just about to start running; recall that thread is leapt forward to VT_s, the time at which the last thread was suspended. When a thread is scheduled it progresses in both real and virtual time. The "stepping" is a feature of running the threads on a single core processor - only one can advance at a time can advance in real time and the threads are alternately scheduled.

Suppose that a test is run to investigate the behaviour of the program if the produces was twice as fast. This is implemented in VTF by giving the producer a TSF of 2. Note that both virtual timeslices remain constant and the two threads are still alternately scheduled. However, assuming a virtual timeslice of 100ms, the producer's real timeslice will now be 200ms, twice the size of that of the consumer (Figure 4). The producer will be allowed to run twice as much code as the consumer each time it is scheduled. This is how VTF

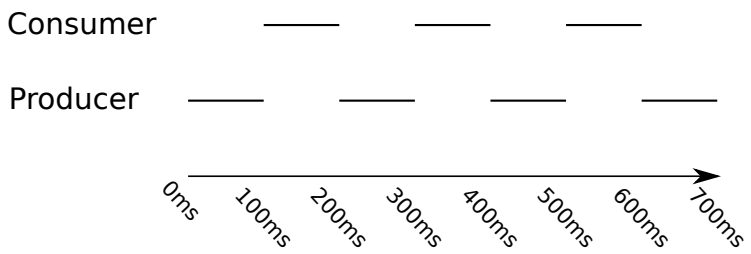

Figure 2: Real time execution trace of threads in the producer/consumer example on a single core with no time scaling. Solid lines indicate a thread being executed.

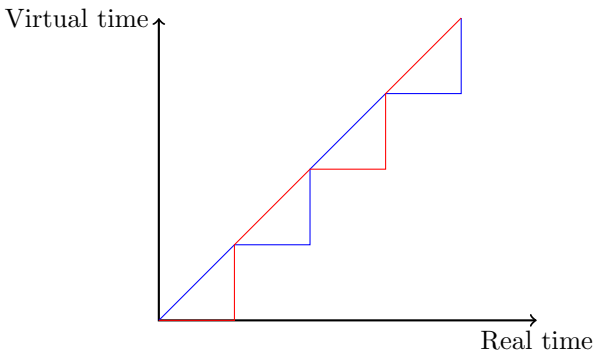

Figure 3: Progress in real and virtual time of the producer (blue) and consumer (red) threads from Figure 2.

simulates accelerated or decelerated code.

We again plot progress on the real and virtual timelines in Figure 5. Note that when the producer thread is run the line representing it (blue) takes a shallower gradient. This is because of the time scaling applied to that thread, which makes the producer thread progress through virtual time at half the rate of real time. In general, the gradient of a line corresponds to the inverse of the TSF applied to the thread in represents.

2.6.5 Multicore Virtual Time Execution

The key problem faced by virtual time executions simulating a multicore processor is very similar to the PDES problem; given multiple virtual timelines, what is the best way to ensure events are processed in the correct order without serialising the system onto one global timeline?

The virtual timelines of the PDES problem correspond to cores in a virtual time execution. To see the issue for VTF specifically, consider again producer/consumer example with a TSF of 2 on consumer, but now both threads run in parallel on a dual core processor. In a simulation of an m core multiprocessor, VTF manages m *local* virtual timelines, so m threads can progress concurrently. Naively applying the scheduling algorithm from single core virtual time execution could lead to each thread running on one core. Given its TSF

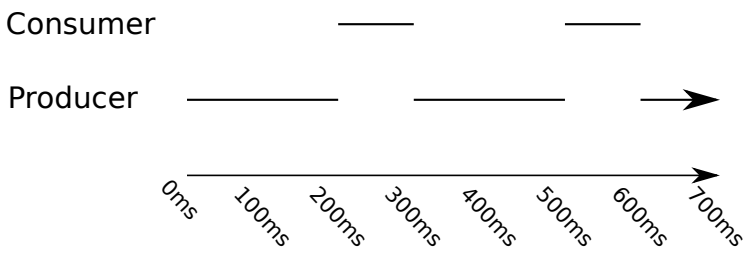

Figure 4: Real time execution trace of threads in the producer/consumer example where the consumer has a time scaling factor of 2. Solid lines indicate a thread being executed.

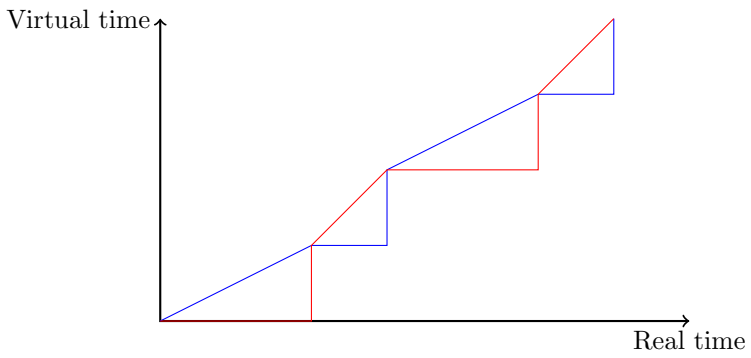

Figure 5: Progress in real and virtual time of the producer (blue) and consumer (red) threads from Figure 4.

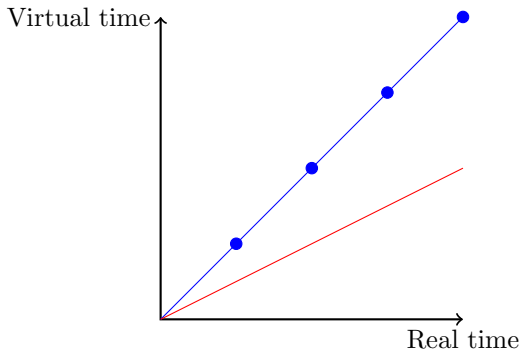

Figure 6: Dual core simulation of the producer/consumer (blue/red respectively) example with time scaling and no synchronisation.

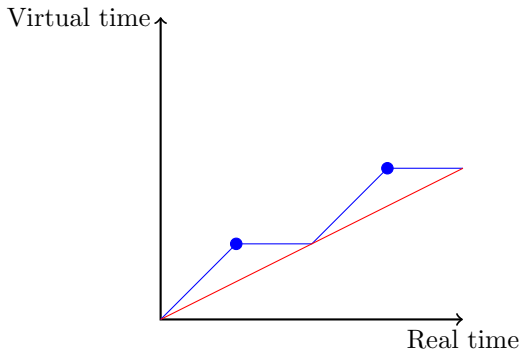

Figure 7: Synchronised dual core simulation of the producer/consumer (blue/red respectively) example with time scaling.

of 2, the consumer will progress through virtual time at half the rate of the producer with respect to real time. The producer will therefore run ahead of the consumer in virtual time. This situation is shown in Figure 6. If the producer places work items on the queue at a constant rate (marked with circles), the consumer will find them on the shared queue at that point in real time. As a result, the producer now see work items appear at twice the original rate in virtual time (since it progresses more slowly) and still not get any additional real time to process them. Recall that the aim of this simulation was to investigate a consumer which was twice as fast - i.e. saw work items enter the queue at the original rate in virtual time but half the rate in real time. Without an inter-core synchronisation scheme, the simulation breaks down.

In order to ensure a correct schedule in virtual time an inter-core synchronisation scheme is necessary. An example of a synchronised dual core simulation is shown in Figure 7. The core running the producer thread is prevented from running ahead in virtual time which allows the consumer sees work units enter the queue at the correct rate. The i-th virtual timeline represents virtual time progress on core i, each with a virtual time VT_i. Multicore synchronisation schemes aim to keep the virtual time progress of these timelines close in real time, while maintaining a low synchronisation overhead.

VTF differs from most PDES systems in that checkpointing/rollbacks are not used. These are expensive for PDES, but in the case of of VTF where program state is arbitrarily large checkpointing would not be feasible.

LAX Synchronisation

The LAX multicore synchronisation scheme assumes that in the absence of time scaling, performance models or leaps forward in virtual time all cores in a simulation will progress through virtual time at similar rates so synchronisation only needs to be applied when one of these conditions holds. This assumption seems reasonable, as virtual time will by default progress at the same rate as real time. Making this assumption reduces the amount of work that needs to be done to provide synchronisation, so should reduce synchronisation overhead.

If one of the special cases holds, for example if a running thread is has a $TSF \neq 1$, synchronisation will be applied. The minimum core virtual time is $VT_{min} = min(VT_1, ..VT_m)$. LAX will generate a barrier to apply the constraint that no core may advance to a virtual time more than $VT_{min} + scheduler\_timeslice$. All running cores are within one scheduler timeslice of the minimum core virtual time. Any cores outside of this interval will be blocked and the simulation threads running on them will not be allowed to progress until the others have caught up.

The LAX synchronisation assumption, that by default all cores progress through virtual time at the same rate, does not always hold. Many things can cause a core to make less progress in real time (and thus virtual time) than its allocated timeslice. For example, background load from other processes running on a simulation host may cause the OS to schedule other tasks instead of the simulation thread for part of the timeslice allocated by VTF. Time spent in the page fault handler is not counted in the CPU time counters which VEX uses to measure core progress, but does consume real time. In such cases, cores will progress through virtual time at different rates with respect to real time, so will fall out of sync.

Strict Parallel Execution

The Strict Parallel Execution (SPEX) synchronisation scheme subjects the simulation to tighter synchronisation bounds. The aim of this stricter synchronisation scheme is to handle circumstances when the LAX synchronisation assumption does not hold. Tighter synchronisation increases the performance overhead of SPEX, but increases its accuracy.

SPEX introduces the concept of *active* and *disabled* cores. A core is active when there are threads to run on it. If no runnable thread exists to be scheduled on an active core it becomes disabled. This distinction is used in the synchronisation scheme. SPEX also defines the *global virtual time* GVT as the minimum virtual time of all active cores in the simulation.

Before resuming a suspended thread on the i-th core, the SPEX synchronisation scheme will apply one of the following rules considering the GVT, the virtual time at core i VT_i and the number of active cores:

- If $GVT + scheduler\_timeslice > VT_i$ core i is permitted to continue execution.
- If $GVT + scheduled\_timeslice < VT_i$ and there are other active cores, core i is not permitted to continue and will idle until the other cores have caught up.
- If $GVT + scheduled\_timeslice < VT_i$ and all cores are disabled there is no other activity in the simulation (as all cores are disabled) so core i is permitted to leap forward to VT_i as nothing could occur in the intervening time.

Following this synchronisation scheme prevents running threads from being more than one timeslice apart.

Chapter 3

Distributed Virtual Time Execution of Programs

The key contribution of this project has been the development of a synchronisation scheme which allows VTF to produce performance predictions for distributed system. Macroscopic synchronisation is provided by enforcing lock step execution of threads in virtual time, similar to SPEX. Messages between processes are timestamped. We find experimentally that is enough to get reasonably accurate results.

3.1 Approaches to Synchronisation

We have previously noted that multicore and distributed synchronisation in VTF is an instance of the PDES synchronisation problem. Given this, we consider the approach we have developed in this project in the greater context of existing PDES synchronisation algorithms.

3.1.1 Conservative Algorithms

Recall that conservative synchronisation algorithms ensure that causality errors never occur, so a process never receives a message to be delivered at a virtual time which it has already passed. We have chosen not to use a conservative approach to synchronisation because ensuring that causality errors could never occur would incur a very high synchronisation overhead.

3.1.2 Optimistic Algorithms

So called "optimistic" synchronisation algorithms bet that allowing a process to run ahead in virtual time will not lead to a causality error. When they are correct they reap the benefits of parallelism between processes. When they are not, they have to deal with a causality error. In PDES this is typically done using checkpointing and state rollbacks.

Checkpointing in VTF would not be feasible as the program state is too large and complex.

Our optimistic synchronisation algorithm differs from the traditional PDES form because we have no way to repair causality errors and must instead accept them into the simulation. From the point of view of the application under test, a causality error will appear as latency; a message will be delivered to the application later than it should have been. This is valid behaviour which could happen in the "real world", for example due to network latency. Introducing additional latency should not break a well engineered distributed system. Allowing causality errors to pass into the simulation therefore does not fundamentally break the simulation (any simulated virtual time execution is an instance of a possible real time execution) but does reduce simulation accuracy.

In a traditional PDES environment, the rate of passage of virtual time bears no relationship with the passage of real time. As a result processes will naturally diverge in virtual time if no attempt is made to provide synchronisation. In contrast, recall from Section 2.6.5 the LAX synchronisation assumption; in the absence of virtual time scaling, performance models or leaps forward in virtual time, all threads in VTF will progress through virtual time at the same rate. The LAX assumption does not always hold, but it is enough to say that threads in VTF will tend to progress together in virtual time. Using this, we argue that causality errors experienced by VTF will generally be smaller than those found in a full PDES system so allowing them to pass into the simulation has less of an effect on accuracy.

3.2 Synchronisation Scheme

We have developed an extension to the SPEX multicore virtual time execution strategy originally proposed by Nick Baltas [6]. This synchronisation scheme should be run alongside SPEX.

We redefine the *global virtual time*, GVT, introduced by SPEX to be the minimum virtual time across all active threads on all nodes of the simulation. Suspended threads which are more than one timeslice ahead of the global virtual time should not be run. If there are no runnable threads in this interval the simulation node should idle.

Applying this throttling mechanism has the effect of keeping the virtual times of all running threads in the interval $[GVT, GVT + 2 * timeslice]$ as only threads in the interval $[GVT, GVT + timeslice]$ will be scheduled, and a scheduled thread will run for at most one timeslice. This approach keeps the amount of global state low; there is only one value, the current GVT, which every node needs to know. Once a node has a value for the GVT it has enough information to decide which of its threads it can or cannot execute. Since the GVT is strictly increasing a node which makes scheduling decisions using a stale value will never incorrectly schedule a thread which it would not have with an up to date GVT (but might not schedule threads which it could do).

All messages sent between simulation nodes should be annotated with the current virtual timestamp of the thread sending them. When a node receives a message, the receiving thread should leap forward in time to the timestamp of the message. We note that this

assumes negligible network latency.

We define the *message delivery error* to be the difference between the actual virtual time a message is delivered to an application and its receive time, as stated by the sender in its timestamp. By following the above synchronisation scheme, the maximum possible message delivery error which can be attributed to VTF is bounded above by $2 * timeslice$. This is because all running threads are in the interval $[GVT, GVT + 2 * timeslice]$ in virtual time, so the receiver could not be more than two timeslices ahead of the sender. We note that a program may experience unbounded message delivery errors, for example due to queuing in the network or if an application does not make a `read` call for a long time, but guarantee that the contribution of VTF to these will be bounded as stated above.

3.2.1 Known Issues

This synchronisation scheme aims to strike a balance between simulation accuracy and overhead, and therefore makes some trade-offs.

Message Delivery Errors

The loose synchronisation for messages introduces bounded delivery errors, but these might still be large enough to effect results. With a 100ms timeslice, the maximum expected error would be 200ms. If errors of this size are common the framework would not be suitable for profiling low latency applications. The delivery error bound can be reduced by picking a smaller timeslice but this means tighter synchronisation between simulation processes so would provide increased accuracy at the cost of increased overhead.

Performance

The global virtual time will advance at the rate of progress through virtual time of the slowest simulation process, meaning that faster processes are held back. We believe this is an unavoidable characteristic of the platform. If faster processes were permitted to go ahead in time higher message delivery errors would be observed when they interacted with a slower process.

Changes to Apparent Network Bandwidth

We expect to find that the apparent bandwidth of a network link between two simulation nodes differs from reality because messages run through the network in real time but are sent and consumed by processes at rates in virtual time. This is similar to the time scaling effects used in network emulation.

Consider two processes which advance through virtual time at half the rate of real time communicating across a single 1Gbps link. The link will become fully saturated when a process sends 1Gb of data down the link in one second of real time. However, in that one second of real time, the process will have advanced half a second in virtual time, so the link

will appear handle data at a rate of 2Gb per second from the point of view of the simulated application.

3.2.2 Necessity of Synchronisation

Consider a system of two processes: a client which sends messages to a server. A test is run simulating a system where the client produces messages twice as fast. Under VTF this would be implemented by decelerating virtual time on the client by a factor of two (applying a TSF of 2). Only virtual time is changed; the client still produces messages at the same rate in real time. If no synchronisation exists between the two processes, the server will therefore still see the messages arriving at the same rate as if the client were not accelerated, only with smaller timestamps.

Under the synchronisation scheme described above, the simulation node for the server will be idle 50% of the time. Because the client's real timeslices are twice the size of the servers, the client takes twice as much real time to advance an equal amount of virtual time as the server. The client will therefore hold back the global virtual time and prevent the server process from running ahead. Note this is almost exactly the same as the multicore example visualised in 7, but with the producer and consumer threads on different machines.

3.3 Implementation

A distributed VTF simulation has a single *master* node which is responsible for coordinating all other nodes, the *followers*. The master is specified at simulation startup using command line parameter passed to VTF. No attempt has been made to provide fault tolerance in the case of master failure or to elect a master dynamically.

On simulation startup all followers will make a TCP connection to the master, forming a star network. The TCP connection will be used by the follower to submit state updates and by the master to provide notifications of global virtual time advances. It is also used by the master to detect follower failures or termination. We note that this channel is used to transport time critical messages between the master and each follower, so it is import that the Nagle algorithm is disabled.

Followers wait for the master node to send a start signal before beginning the simulation. This is to ensure all followers have time to connect to the master and reduce the effect of staggered starting times during the experiment setup. At the time of writing the master waits for 10 seconds before sending this signal, but could alternatively wait for a specified number of followers to connect.

3.3.1 Macroscopic Synchronisation

This section describes the implementation details of the synchronisation scheme which keeps running threads in the interval $[GVT, GVT + 2 * timeslice]$. The global virtual time is computed by the master based on candidates submitted by followers. Changes to its value are

relayed to the followers. Each core of a simulation process enforces the constraint that no newly scheduled thread may start running at a virtual time greater than $GVT + timeslice$, which keeps all executing code in the above interval.

Candidate Virtual Times

Followers regularly submit updated candidate virtual times to the master. The candidate virtual time will be the minimum virtual time of all the runnable (VEX state running or suspended) threads on this host. Candidate generation and submission is handled by a dedicated thread to avoid performing network I/O in code executed by instrumentation.

Calculating the GVT at the Master

The global virtual time is the minimum of submitted candidate virtual times from each follower, excluding times from simulations which have no runnable threads and have been blocked on network IO for more than a certain duration of real time, $min\_block\_duration$. These processes are, for example, blocked on network reads, so should not prevent other simulation processes from making progress.

The value of $min\_block\_duration$ is selected so once it has expired we can be sure there is no message currently in the network which will awaken the blocked process. At this point, the only way for the simulation to make progress is to allow other simulation processes to advance.

To see why excluding the candidate from a blocked process is necessary, consider a system of two processes: a client and server that communicate over a network. The client will submit a request to server and immediately block on a network read. Its candidate virtual time will not advance until it exits this read. What happens is shown in Figure 8. If the global virtual time includes client's candidate and the server takes more than two timeslices to generate its response, the system will deadlock. This is because the the client holds back the global virtual time, so the furthest the server can advance is $GVT + 2 * scheduler\_timeslice$ (black dashed line in Figure 8. If the server takes longer than this to respond it will be suspended pending advances in the GVT. The client will not advance in virtual time until it receives a response from the server. We therefore reach a deadlock, where the server will not be able to advance until the client does and vice versa.

The solution to this deadlock is to allow remove the client's virtual time from the GVT computation. With the clients candidate virtual time excluded the global virtual time will be able to advance beyond the virtual time of the client. When the server responds to the client, the message timestamp will be in the interval $[GVT, GVT + 2 * timeslice]$, so the client will return to the valid runnable interval upon receipt of the message.

3.3.2 Additional Instrumentation

We have extended JINE to statically instrument the `java.net.Socket` and `java.net.ServerSocket` classes which provide stream sockets to programs running on the JVM, representing active

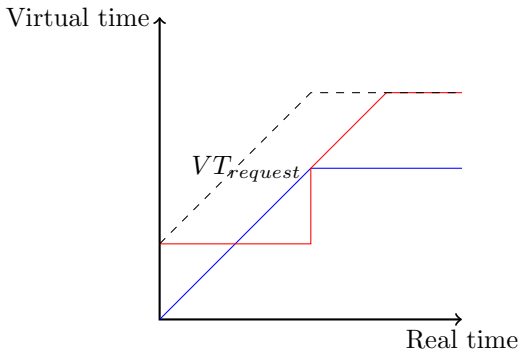

Figure 8: Potential deadlock if a blocked client is allowed to hold back the GVT. Client in blue, server in red. The black dotted line is the $GVT + 2 * scheduler\_timeslice$.

(connecting) and passive (accepting) sockets respectively. JINE also dynamically replaces calls to `ServerSocket.accept` in application code at runtime.

VTF will not be aware of any network I/O that does not rely on these classes. This means VTF will not properly annotate messages sent using the `java.nio.channels.SocketChannel` class, or native libraries. It would not be difficult to add instrumentation to the `SocketChannel` class.

Instrumentation Level

JINE adds instruments by altering the bytecode of certain system and application classes. Working at the this level has advantages and disadvantages. Adding instruments in this fashion matches how the rest of JINE works, so the framework required by our extensions already exists. Conceptually, we want to annotate each message sent between simulation nodes. However, sockets provide an interface based on streams, not messages. A programmer makes `write` calls which cause messages to be sent, but a particular `write` may result in multiple TCP packets being transmitted, or none at all if the Nagle algorithm is in use. At the receiving host the data collected by a `read` does not necessarily correspond to a particular `write`; it may be collected from many small `write`s or be only a fraction of the data from one large one. The discrete messages (TCP packets) which are passed between simulation nodes are hidden by the socket API, so we instead have to insert annotations into the TCP bytestream upon a `write` and scan for the annotations during a `read`.

Alternatively instrumentation could be inserted at a lower level. Placing instrumentation within the network stack would have the advantage of being able to timestamp each individual TCP packet rather than inserting annotations into the connection bytestream. The would be a much larger engineering effort and would not fit with the current VTF philosophy of avoiding kernel level code. The internals of the network stack are also much more likely to change than the `Socket` and `ServerSocket` classes, so patching it would introduce additional maintenance costs.

Placing instrumentation at a higher level is also an option. Instruments could be manually placed in particular libraries, e.g. in HTTP client/server libraries. This is the approach taken by Google in Dapper [12]. In their case it is appropriate since they can identify a small number of libraries which need to be instrumented in order to capture most interactions between elements in their target applications. This approach is not fit for purpose in VTF as the framework aims to be general.

`bind` method of `ServerSocket`

The `bind` call signals that the OS should begin to accept TCP connections on a specified port [1]. These connections will be placed in the backlog queue until an `accept` call from the application code requests them. Our instrumentation for `bind` calls starts a simulated backlog queue (see below).

`connect` method of `Socket`

A `connect` call attempts to establish a TCP connection to a remote host. We have added instrumentation which makes calls into VTF before and after a thread makes a connect call. Instrumentation before the blocking call will update the thread's state within VTF to reflect its being blocked. When the call returns, further instrumentation will revert this state change and send this thread's current virtual time down the newly established channel. This tells the remote host what the virtual time is at the connecting host (VT_C), and therefore the earliest virtual time at which that application code should receive this connection.

`accept` method of `ServerSocket`

An `accept` call takes a queued TCP connection from the backlog queue, or if the queue is empty blocks until one is established. JINE replaces calls to `accept` in application code with a call into VEX which picks the connection from the head of the simulated backlog queue and leaps the thread forward to the virtual time that connection was established. After the call returns further instrumentation code will read the virtual time which was written by the `connect`ing thread from the channel and leap the thread forward to that time.

Figure 9 shows the connect/accept synchronisation mechanism in action. The blue line represents a thread blocking in `accept` at VT_A, after which it does not advance in virtual time as real time passes. When another thread (represented by the red line) `connect`s at VT_C the accepting thread unblocks and is leapt forward to VT_C.

It is important to note that a `connect` call returns when a TCP connection has been established which, from the application's point of view, only requires that a socket is bound to this port on the accepting host. It does not necessarily mean that a matching `accept` has returned on the remote host. This is reflected in the asymmetric flow of virtual times, i.e. only from the `connect`ing process, during socket connection.

[1] Many socket implementations use a `listen` call to mark a socket as passive and start queuing connections, but the JDK makes this differentiation at the class level (`ServerSocket` is passive, `Socket` is active) so the `listen` is implicit in a `bind` on a `ServerSocket`.

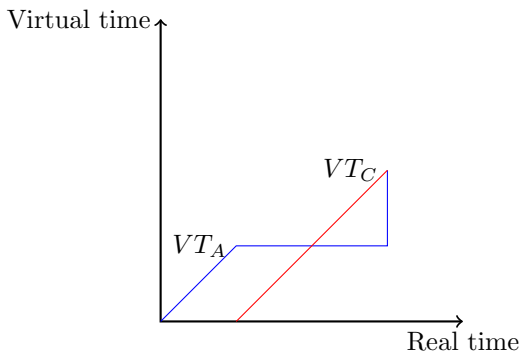

Figure 9: Interactions in virtual time for a thread calling `accept` at VT_A (blue) and a thread calling `connect` at VT_C (red).

`read` and `write` methods of `Socket`

Instrumentation of the `write` method is used to add timestamp annotations to messages. The virtual time of the current thread is inserted into TCP byte stream, marked out by a special pair of marker bytes. The marker is written to the socket followed by the sending thread's current virtual time. Instrumentation of `read` calls scans the receive buffer for the marker. When it finds a marker, the instrumentation code will remove it and the subsequent 8 bytes (the virtual time) from the buffer before returning it to the application. The receiving thread is leapt forward to the annotated time.

Reordering the backlog queue

Since nodes in a distributed VTF simulation are loosely synchronised, the order in which connections are established in real time is not necessarily the correct order in virtual time. Consider a hypothetical single threaded "time" server with two clients. The server accepts connections sequentially and responds with the current time. For simplicity we assume that the server is infinitely fast. Clients initiate connections and immediately block in a read. Suppose client A connects first in real time, but is ahead of client B in virtual time as visualised in 10. $VT_{C1} > VT_{C2}$ so the connection from client B happens first in virtual time. However, taking connections in the order they come from the OS provided backlog queue would mean client A would be serviced first as it made its connection first in real time. Notice that the server leaps forward in virtual time as soon as client A connects. This introduces inaccuracies in the simulation by causing client B to experience an increased service time as the response from the server will be from a virtual time *after* the response for client A despite its request being made at a virtual time *before* that of client A. The real time ordering causes the incorrect leap forward in virtual time shown in 10 after VT_{C2}.

To remedy this, we simulate the backlog queue ordering connections in ascending virtual time order, rather than real time. Each time a `listen` call is made (which occurs in the `bind` method of the `ServerSocket` class) a new thread is spawned to accept connections and place sockets in the simulated (virtual time ordered) backlog queue. `accept` calls in the application take sockets from this queue. If the socket at the head of the simulated

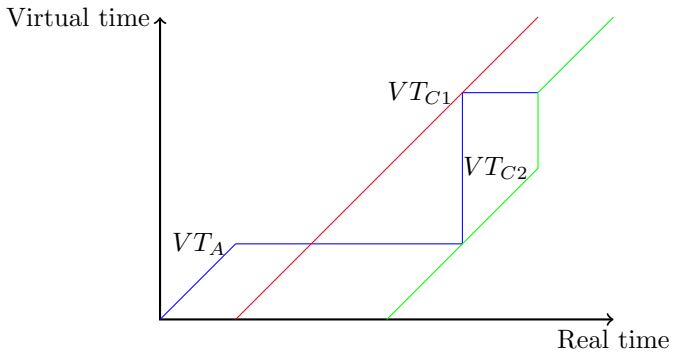

Figure 10: A server (blue) accepts connections in real time order from clients (red and green), leading to an increased service time for the green client.

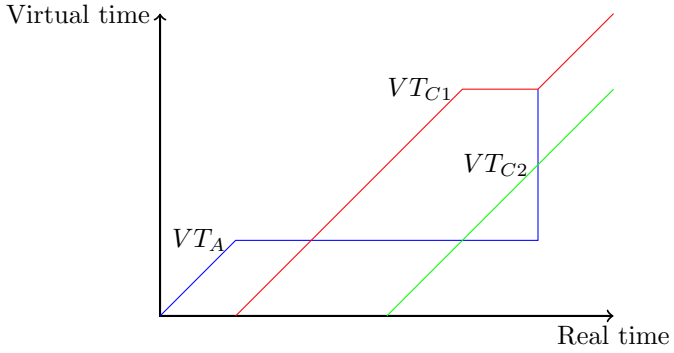

Figure 11: A server (blue) accepts connections in virtual time order from clients (red and green), with "normal" service times for the both clients.

backlog queue is far forward in virtual time when an accept call is made, VEX will block the accepting thread until the GVT has advanced close to it or a timeout has passed. This aims to reduce the probability of prematurely leaping forward to accept a connection when other connections may come in at earlier virtual times. The timeout is in place to reduce the performance overhead of imposing a stricter synchronisation scheme and preclude the possibility of deadlocks. We note that it is still possible for connections to be handled out of order, but less likely.

Figure 11 shows how simulating the backlog queue fixes the problem of out-of-order connections in virtual time which was described above. The server application will remain blocked in the `accept` until the GVT has advanced close to the time at head of the simulated backlog queue. This means that client A's connection is not immediately passed to the application at VT_{C1}. As a result, client B has enough time to make its connection, which will jump in ahead of client A in the queue as client B is at an earlier virtual time. Now, client B is serviced before client A, so both see normal response times.

Chapter 4

Experimental Results and Analysis

In this chapter we present results of running experiments using simple programs to demonstrate the distributed synchronisation scheme we have developed in action. We show its effectiveness at keeping processes synchronised in virtual time and how it allows VTF to make performance predictions for distributed applications. We also analyse the performance overhead and scalability of our approach.

We can only expect the results of our tests to be as accurate as the underlying framework that produces them. Prior evaluations of VTF have found it to be fairly accurate, with a mean prediction error of 8% in the SPECjvm2008 multicore benchmark [6] using SPEX with no time scaling. In the same benchmark simulation overhead was measured at 30%.

4.1 Analysis of Connection Errors

In these tests we show experimentally that the maximum error introduced by VTF at the connection stage is small (typically much less than the $2 * scheduler\_timeslice$ limit). We demonstrate that it remains so under time scaling and is reduced when the scheduler timeslice is reduced.

Our tests use a single server which serves client requests sequentially on a single thread, pseudocode for which is shown in Figure 12. We emphasise that server will handle requests as fast as it can by immediately accepting a connection, writing the current date to it and immediately closing the socket. All tests use 50 clients (Figure 13). These clients make requests to the server by opening connections and reading the message sent from the server. In between requests to spin in a "timewasting" function. This "think time" at the client is exponentially distributed, with rate parameter λ selected such that load on the server is very low.

We run experiments to measure the difference between the server thread's virtual time when in blocks in the `accept` VT_A at line 2 of Figure 12 and the virtual time of the client

thread which connects next VT_C. We call this value, $VT_C - VT_A$ a *connection error*. As server utilisation is so low the server will spend most of its time blocked in an `accept` call, so the backlog queue will almost always be empty. In a fully correct simulation we expect $VT_C - VT_A > 0$. This is the desirable case; the server thread will leap forward to VT_C to handle the request and the client thread will experience a normal service time. If $VT_C - VT_A < 0$, a "negative leap forward", the server thread has already passed VT_C, when the connection should have been made, causing the client to experience a larger than usual service time as it includes an error of $VT_A - VT_C$.

The connection error is the parallel of the *message delivery error* discussed in Section 3.2 in terms of `connect` and `accept` calls rather than `write` and `read` calls. In this test the dynamics of the message delivery errors are quite straightforward. The server never makes a `read` call, so receives no messages in this sense. The client blocks immediately in a `read` after establishing a connection to the server, which should respond immediately, so its message delivery errors are almost exclusively zeros.

```
1 while more requests are expected do
2 |   accept a connection yielding socket s;
3 |   write the current time to s;
4 |   close s;
5 end
```

Figure 12: Server application code for the connection delay test.

```
1 for i in 1 .. numberOfRequests do
2 |   open a connection to the server yielding socket s;
3 |   read from s;
4 |   close s;
5 |   do work taking time t ~ exp(λ);
6 end
```

Figure 13: Client application code for the connection delay test.

4.1.1 Effect of reducing the scheduler timeslice

As stated in Section 3.2, our synchronisation scheme holds all running threads in the interval $[GVT, GVT + 2 * scheduler\_timeslice]$. Reducing the scheduler timeslice means that threads will be more tightly synchronised so we would expect to see reduced connection errors.

Since server utilisation is low, the server will spend most of its time blocked in `accept`s and will leap forward in virtual time when a client connects. Progress in virtual time of the server is driven by requests coming in from clients. Errors are introduced when connections are handled out of order in virtual time. This can only occur when two clients pick connection times within two scheduler timeslices of each other (if they were further apart the connections could not happen out of order). Reducing the timeslice reduces the prevalence

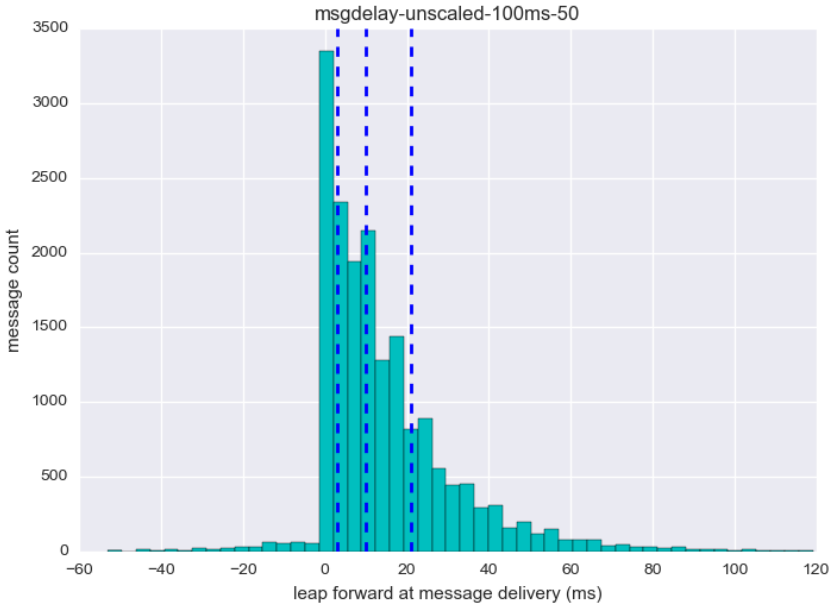

Figure 14: Histogram of $20,000$ connection errors, $TSF = 1$, $scheduler\_timeslice = 100ms$

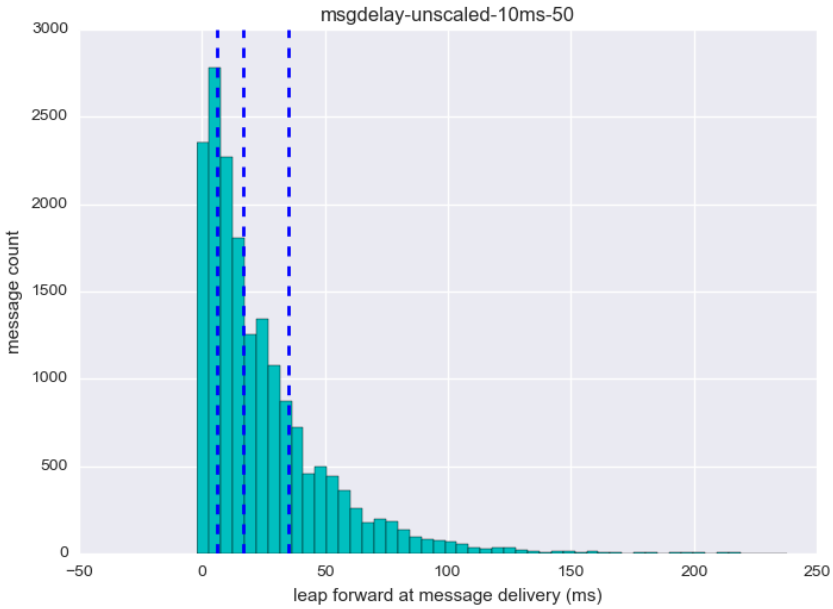

Figure 15: Histogram of $20,000$ connection errors, $TSF = 1$, $scheduler\_timeslice = 10ms$

of errors by making it less likely that two clients will pick connection times which could be run out of order in real time.

Figures 14 and 15 show measured values of $VT_C - VT_A$ from experiments with timeslices of 100ms and 10ms respectively. In these Figures and those which follow, the three blue vertical lines on this histogram denote the $25th$ percentile, the median, and the $75th$ percentile. We note that in a fully correct simulation these histograms would show the interarrival time of requests at the server.

With a timeslice of 100ms approximately 2.8% of connections lead to a negative leap forward. Reducing the timeslice to 10ms cuts this down to less than 1%. Reducing the timeslice tightens the synchronisation between threads, so reduces the number of client requests to the server which could be handled out of order in real time, as these must occur within 2 timeslices of each other in virtual time. This accounts for the reduction in negative leaps forward.

Recall that the client think times are distributed as $exp(\lambda)$. Since the service time at the server is almost 0, clients will spend almost all of their time thinking so the distribution of arrival times at the server will be approximately $exp(50\lambda)$. Indeed, the shapes of the histogram in Figures 14 and 15 match what we would expect for an exponentially distributed random variable.

4.1.2 Effect of Time Scaling

Next we consider the effect of applying time scaling to the "timewasting" function. The client code is unchanged but has a TSF applied to the timewasting code. This changes the client think time and therefore the rate at which clients send requests to the server. Applying a TSF of 2 will halve the think time at the client and therefore double the rate at which the connections arrive at the server. In these experiments we have used a 100ms timeslice.

Negative Leaps Forward

First we consider Figures 16 and 17, in which the client is accelerated in virtual time by applying a TSF of 0.5 and 0.2 respectively. We note the reduced number of negative leaps forward, detailed in 21. A TSF of less than 1 means that the the client think time is increased and therefore reduces load on the server. Client requests are therefore spread over a larger period of virtual time but still synchronised to within the same $2 * timeslice$ interval. A negative connection error will occur when two requests are processed out of order. For this to happen both must occur within the same 2 timeslice interval as the synchronisation scheme does not allow threads to diverge by more than this. Spreading client requests over a larger period reduces the likelihood of two requests falling in the same $2 * timeslice$ interval. Requests are therefore less likely to be processed out of order, leading to a reduced prevalence of negative leaps.

Figures 18 and 19 show leaps forward for experiments with a TSF of 2 and 5 respectively. A TSF greater than 1 simulates clients which are faster than reality, with smaller

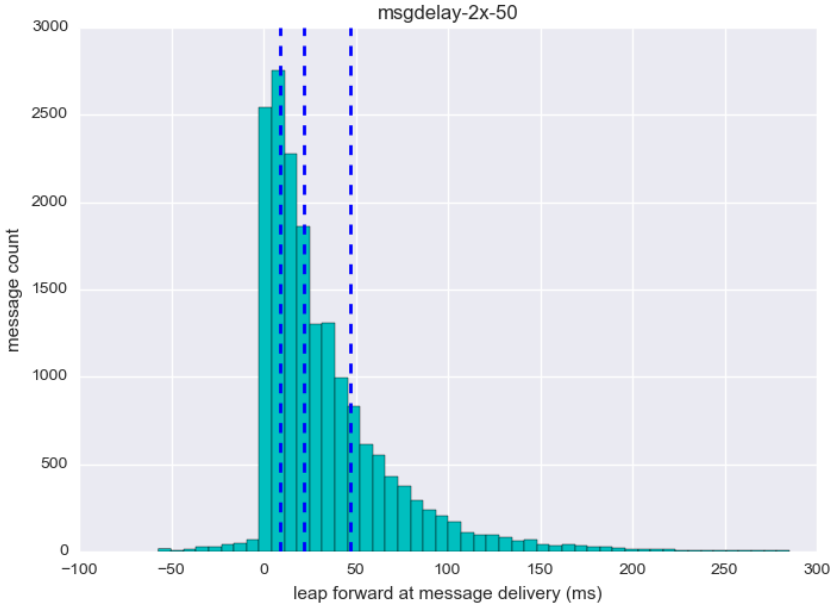

Figure 16: Histogram of $20,000$ connection errors, $TSF = 0.5$, $scheduler\_timeslice = 100ms$

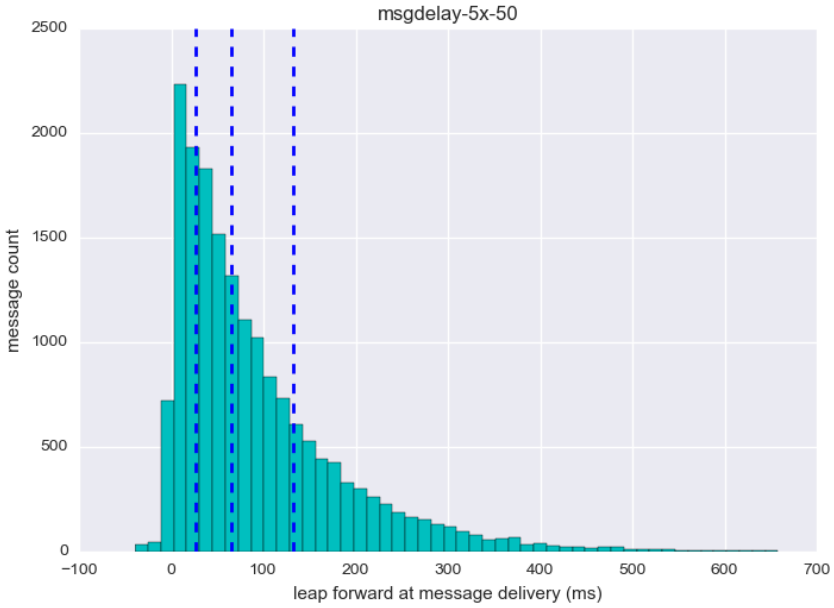

Figure 17: Histogram of $20,000$ connection errors, $TSF = 0.2$, $scheduler\_timeslice = 100ms$

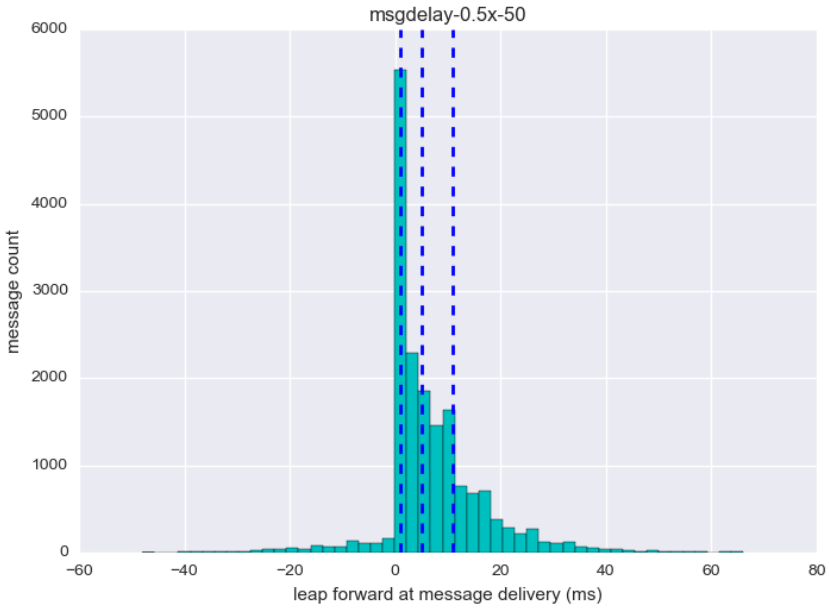

Figure 18: Histogram of $20,000$ connection errors, $TSF = 2$, $scheduler\_timeslice = 100ms$

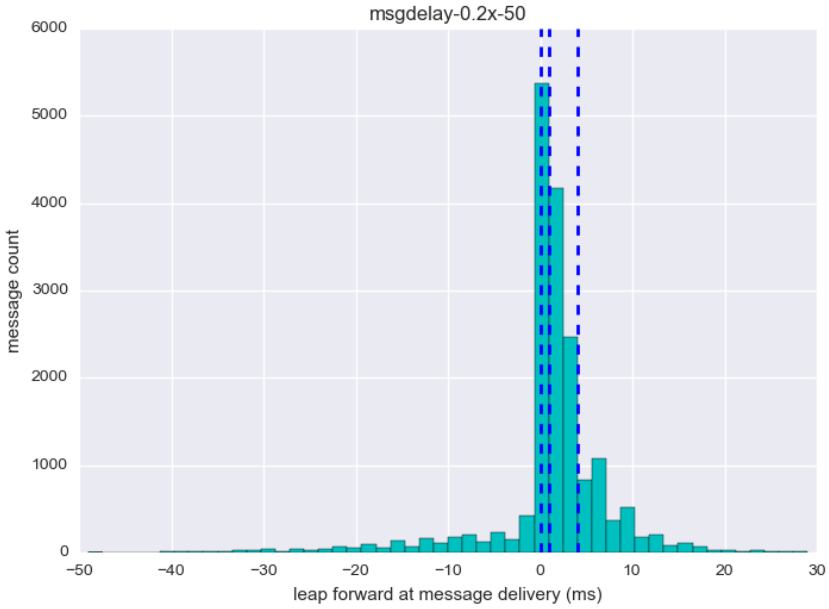

Figure 19: Histogram of $20,000$ connection errors, $TSF = 5$, $scheduler\_timeslice = 100ms$

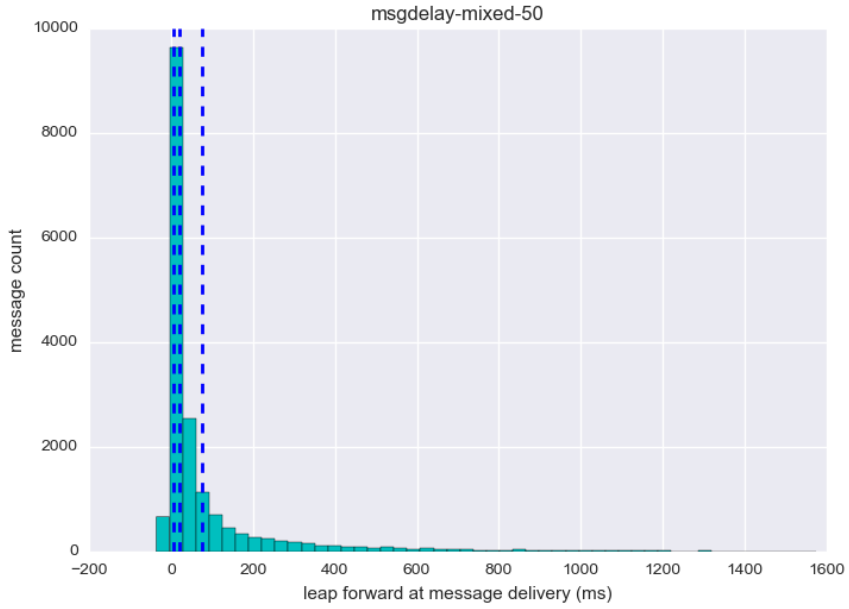

Figure 20: Histogram of 20,000 connection errors, mixed client TSF, $scheduler\_timeslice = 100ms$

think times between requests. Negative leaps forward become more common as detailed in Figure 21, increasing from 2.8% of cases in the unscaled experiment to 5.6% and 13.1% of cases in experiments with a TSF of 2 and 5 respectively. This is the same effect as observed in Figures 16 and 17 but in reverse; client requests are compacted into a shorter period of virtual time so are more likely to fall into the same 2 timeslice and still synchronised to within a $2 * timeslice$ interval, so are more likely to be processed out of order. The leads to more negative leaps forward at the server.

In Figure 20 we show the leaps forward upon connection when the clients have a variety of time scaling factors applied to them. This test has 10 clients with a TSF of 1, 2, 5, 0.5 and 0.2, a total of 50 clients. Observe that despite the rates of progress in virtual time of some clients differing by a factor of 10, the left hand tail of negative leaps forward remains small. 5.8% of connections are handled late.

We note that in all cases the worst (most negative) connection errors remained much smaller than the maximum possible error of $2 * scheduler\_timeslice$.

Movement of Median Leap Forward

We also note the movement of the median leap forward. In Figure 16 applying a TSF of 0.5 causes the average client think time to double, so we would expect the average interarrival time at the server to be doubled. Since message arrivals cause the server to leap forward, we

TSF	median leap forward (ms)	expected	percentage difference	negative leaps
1	10	baseline	-	2.8%
2	5	5	0%	5.6%
5	1	2	-50%	13.1%
0.5	22	20	10%	1.8%
0.2	65	50	30%	1%

Figure 21: Measured and expected median leaps forward under time scaling.

should expect to see the average leap forward double. In general, we predict the median leap forward to be the baseline (unscaled) median divided by the TSF applied in a particular test.

Referring to Figure 21 we see that this prediction is accurate when the TSF is close to 1. The large percentage error for a TSF of 5 has two causes. Firstly, measurements are made at millisecond granularity, so are not really precise enough in this case. Secondly, this experiment exhibits an increased number of negative leaps forward. These causality errors would, in a fully correct simulation, be positive leaps. Instead they sit below zero and pull the median down.

4.2 A simple queuing example, with time scaling

In this section we evaluate the ability of VTF to make performance predictions across multiple machines using the synchronisation scheme we have developed. We do this by measuring the response time of a server under increasingly heavy load from up to 50 clients. We then apply time scaling to the server using VTF to predict the response time were it to handle requests twice as fast. This prediction is shown to be accurate to within the VTF profilers error.

4.2.1 Experimental Setup

The key difference between the programs used in these experiments and the server and client programs used in the previous section is that the service time of the server is no longer negligible. The server will handle requests sequentially, on a single thread. The time taken for the server to generate a response is distributed exponentially with rate parameter λ_S (line 3, Figure 22). As before, the client think time between requests is exponentially distributed with rate parameter λ_C (line 5, Figure 23). We note that the experiment can be modelled as a closed queuing network with an M/M/1 server.

We select $\lambda_S = 20 * \lambda_C$, so expect to see the response time degrade when more than 20 clients are active.

In order to test VTF's predictive accuracy we run same experiments with a TSF of 2 applied to the server's response generation code. This simulates a server which is twice as fast.

These experiments are run with a 100ms timeslice.

```
1 while more requests are expected do
2 |   accept a connection yielding socket s;
3 |   generate response taking time t ~ exp(λ_S);
4 |   write the response to s;
5 |   close s;
6 end
```

Figure 22: Server application code for the response time test.

```
1 for i in 1 .. numberOfRequests do
2 |   open a connection to the server yielding socket s;
3 |   read from s;
4 |   close s;
5 |   do work taking time t ~ exp(λ_C);
6 end
```

Figure 23: Client application code for the response time.

4.2.2 Experimental Results

Figure 24 shows the results of our first pair of experiments, in which we compare response times reported by VTF when no time scaling is applied to real response times from the uninstrumented program. This is for validation purposes; we should see the same performance characteristics in both cases. The blue line shows mean response times for our server under load from an increasing number of clients when run under VTF with no time scaling. The green line shows results of the same test run as pure Java without VTF.

In this test, VTF tends to slightly overpredict the response time. The source of the overprediction is VTF prematurely allowing the server to accept a connection from a client which is far forward in virtual time, leaping the server thread forward in time and causing subsequent connections from earlier virtual times to be handled late. Recall that our simulation of the backlog queue is not perfect, and will will occasionally allow premature leaps forward. These clients therefore experience larger than normal response times, which leads to the overprediction. The same results are tabulated in Figure 26. We see a maximum prediction error of 9% and an average of 5%. These errors are within the accuracy of the VTF framework, so are within the range that we should expect.

The point of the VTF project is to make performance predictions, so we next investigate how accurately VTF can predict the response time of a the single threaded server under time scaling. Figure 25 shows a predicted response time curve for the server were it to serve requests twice as fast (blue line). This prediction is implemented by applying a TSF of 2 to the timewasting function which dictates the servers think time. We see that the response time is much lower than for the original server. The is because of the lower service time

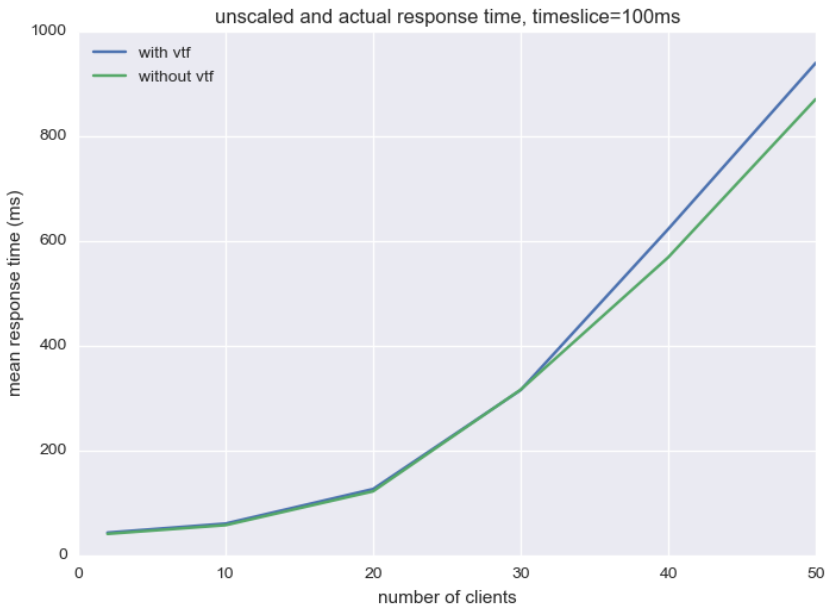

Figure 24: Response time on a single threaded server, no time scaling.

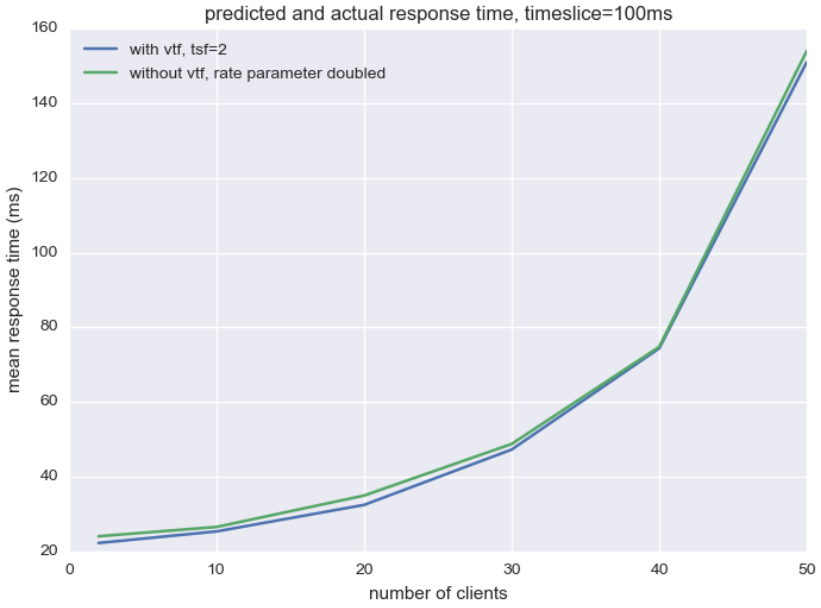

Figure 25: Response time on a single threaded server, with time scaling.

Number of clients	Predicted response time	Actual response time	Prediction error
2	43.6	41.3	6%
10	60.9	57.9	5%
20	126	123	2%
30	316	316	0
40	622	569	9%
50	940	871	8%

Figure 26: Predicted and actual response times for the server with no time scaling in Figure 24. All measurements are means and units are milliseconds.

Number of clients	Predicted response time	Actual response time	Prediction error
2	22.3	24.1	−7%
10	25.4	26.6	−5%
20	32.5	35.0	−7%
30	47.3	48.8	−3%
40	74.4	74.8	−0.5%
50	151	154	−2%

Figure 27: Predicted and actual response times for the server with time scaling in Figure 25. All measurements are means and units are milliseconds.

leads to less queuing.

Since the service time of the server is under our control, we can validate the predicted response time of a double speed server by running the same test in pure Java with a server which is actually twice as fast. Doubling the rate parameter of the service time distribution halves the average service time (green line in Figure 25). We see that the prediction lines up with reality very well.

The results are again tabulated in Figure 27. In this set of experiments VTF underpredicts the response time. The mean prediction error is −4%, which is within the framework error.

4.3 A Multithreaded Server

Our next set of experiments attempts to tackle two ways in which our previous experiments have been unrealistic.

Firstly, a real application would not use a single thread to serve requests. We introduce a more realistic (though still inefficient) server architecture - spawning a new thread to handle each connection.

The second issue is the lifetime of a connection to the server. A new connection is established each time a client makes a request. In this test the client instead establishes a single connection to the server which it uses for multiple request/response cycles for the entire

duration of the test.

4.3.1 Lightly Loaded Server

Our first test uses the same client and and server think time distributions as the single threaded tests in Section 4.1. We note that while we no longer expect to see client requests queuing to connect, the threads which handle them will still queue for CPU time.

Figure 28 shows the response time of this server for an increasing number of clients. In this test, each client makes 1000 requests to the server, the data point for 50 clients is the mean of $50,000$ response times. The response time remains roughly constant because the threaded server architecture takes full advantage of its host's multicore CPU (4 physical, 8 logical cores). This means that the load placed upon the server by this test is very light.

For a single client VTF predicts the mean response time to an accuracy within the framework error. With larger numbers of clients we observe significant prediction errors, increasingly large as the number of clients rises, where VTF predicts the response time of the multithreaded server to be larger than it really is. With 50 clients the error is as much as 126%.

Figure 30 compares the mean service times measured from the tests with and without VTF for an increasing number of clients. As the number of clients increases, the mean service time simulated by VTF increases much faster than the real mean service time. Observe that for a single client the service time is simulated accurately, which allows VTF to accurately predict the response time with a single client.

The multithreaded server has one thread for each client, so in the test with a single client the server only has one thread. When more clients are added, the server will spawn more threads and make full use of the multicore processor simulated by VTF. Since the errors in service times arise when the server has multiple clients, we can place the cause of this problem on VTFs multicore simulation. It appears that VTF does not accurately simulate processor sharing between threads in this multicore environment.

4.3.2 Increased Server Load

We next consider a moderately heavily loaded multithreaded server. We implement this test by increasing the server think time by a factor of 4. Client think times are unchanged.

Response time curves for between 10 and 50 clients are shown in Figure 29. For small numbers of clients (fewer than 30) the prediction is fairly accurate. Past 30 clients, the load increases and we find that VTF does not accurately simulate the real program behaviour. At 50 clients the prediction error is 61%.

Figure 31 summarises the median client think time, service time and response time for this simulation with and without VTF. Observe that the measured client think times are

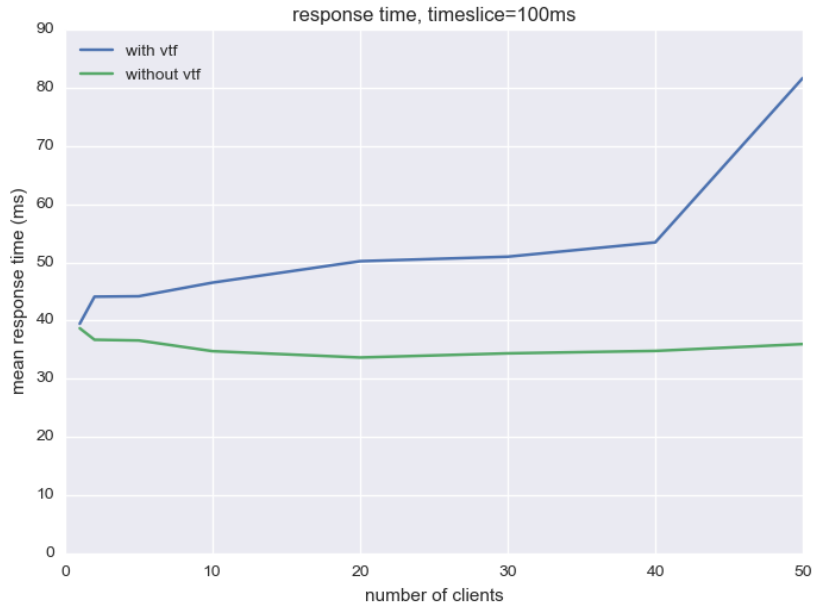

Figure 28: Mean response time on a multithreaded server with light load.

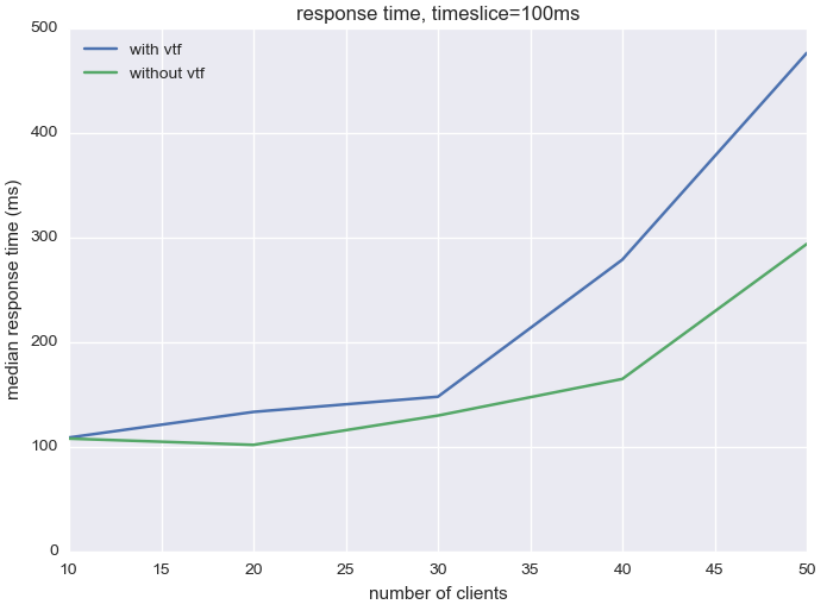

Figure 29: Mean response time on a multithreaded server under increased load.

Number of Clients	Service time with VTF	Service time without VTF	Percentage Difference
1	38.9	37.8	2.9%
2	43.0	35.9	20%
5	43.2	35.9	20%
10	46.1	34.1	23%
20	49.7	33.0	51%
30	50.4	33.8	49%
40	52.3	34.2	53%
50	61.1	35.4	75%

Figure 30: Comparison of service times with and without VTF from the tests in Figure 28. All measurements are means and in millisecond units.

Measurement (ms)	With VTF	Without VTF	Percentage Difference
Client think time	505	461	9%
Service time	433	290	49%
Response time	474	294	61%

Figure 31: Medians measured from the moderately loaded multithreaded servers in figure 29

within the error introduced by VTF. The increased response time reported by the VTF simulation can be attributed to increased service time in the simulation. This error appears to be introduced by VTF in moderate to heavily loaded multicore simulations since it appears only in the multithreaded server's service time, but not the in the single threaded client's think time.

4.3.3 Accuracy of VTF on Multicore platforms

We have continued to diagnose the root cause of this error by devising a multicore stress test similar to the heavily loaded multithreaded server. The test runs on a single node, spawning n threads which each perform a specific number of calls to an exponentially distributed "timewasting" function in sequence, measuring execution time for each call. These tests were executed on an i7 4470 processor with 8 logical cores backed by 4 physical cores using hyperthreading. Simulating 8 cores is the closest approximation VTF can provide to this. Figure 32 shows the median think time spent in each call to the "timewasting" function as the number of threads n is increased from 1 to 50. We see that VTF is only accurate at low levels of load. As processor utilisation approaches 100%, around 6-8 threads, the predicted median think time from VTF diverges from reality. VTF significantly underpredicts the median runtime of the timewasting function for heavier loads that 30 threads.

Figure 33 summarises the distribution of samples for 50 threads in three cases: with full VTF, VTF with distributed extensions disabled and pure Java (without VTF). We note that while the medians differs somewhat between full and non-distributed VTF, the rest of the distribution is very similar. The distribution of samples measures without VTF is somewhat different, with a higher median and a lighter upper tail. From these results we can conclude that the cause of the error in service time seen in Figure 29 likely lies with how VTF treats

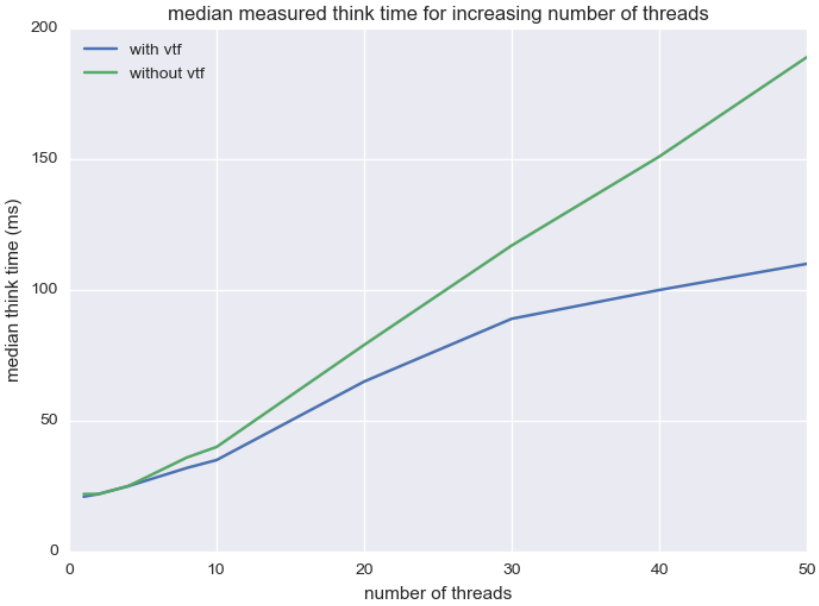

Figure 32: Median think time in "timewasting" function

multicore simulations. Since the focus of this project is distributed synchronisation we leave this problem open as future work.

4.4 A real application

A long term aim of the Virtual Time Framework is to be able to profile real world applications. We have tested our extensions to the VTF framework in using the Jetty servlet container, aiming to assess how close VTF is to this goal.

This experiment consisted of a 50 clients making HTTP requests to a single server. The

Percentile	VTF	VTF, distributed extensions disabled	No VTF
1	0	0	0
10	5	5	6
25	16	15	49
50	107	86	187
75	463	463	410
90	1109	1107	703
99	1809	1857	1444

Figure 33: Distribution of sample "timewaster" runtimes (ms) measured in multithreaded stress test.

server was implemented as an embedded Jetty instance containing a Java servlet which generated responses. As in previous experiments, response generation is modelled using a exponentially distributed amount of work in a "timewasting" function selected so that load on the server is low. By default, Jetty uses non-blocking I/O via classes from the `java.nio` package. Network I/O via these classes bypasses our instrumentation, so we configured Jetty to use blocking I/O and the `Socket` and `ServerSocket` classes. The scheduler timeslice was 100ms. Reported results are averages over 3 runs.

VTF predicted the mean response time of the server to be 68ms. In reality it was measured as 52ms, an error of 31%. We measured a mean service time of 47ms with VTF and 49ms in reality. In this case, VTF did accurately simulate processor sharing on the server. This agrees with the results in Section 4.3, where VTF simulations of multicore platforms are shown to be accurate when few threads are running. Since load on the server is low, the true response time of the server (52ms) is close to the service time (49ms). VTF introduces an error which drives the simulated response time of the server up.

The simulation overhead was measured to be approximately 22%, within the 30% overhead typically shown by VTF on single node platforms. Based on analysis of the client's execution traces, we estimate that around 5% of the overhead is due to the distributed synchronisation scheme.

4.5 Simulation Overhead and Scalability

In this section we investigate the scalability the distributed extensions to VTF. In particular, we consider the effect of two parameters on wallclock simulation runtime: increasing numbers of simulation nodes and decreasing timeslice size.

4.5.1 Number of Simulation Nodes

We expect the global virtual time update mechanism at the master to be the bottleneck in our synchronisation scheme. If GVT updates are not computed and distributed quickly enough simulation threads will be held back by the synchronisation scheme when they could run without introducing errors. The key parameter which controls stress upon the GVT update mechanism is the number of nodes in the simulation; with more nodes, each update computation must consider more candidates virtual times and an updated GVT must be distributed to more followers.

To test this, we have devised a experiment to measure the change in synchronisation overhead as more nodes are added to a simulation. The test program executes a (large) fixed number of iterations in an exponentially distributed work function similar to the "timewasting" functions used in the previous sections. We measure the real time taken for the simulation to terminate. There is no direct communication between the simulation nodes as this would complicate the dynamics of the simulation time (a server would experience increasing load and therefore exhibit increasing response times).

Figure 34 shows the results of running this experiment with a timeslice of 100ms for up to 200 clients. Runtime is allocated into two buckets according to how much time the simulation node spent suspended due to the distributed synchronisation scheme or running. The area under each graph is coloured accordingly. Time spent "running" represents real time spent either running the application or in VTF instrumentation code. The "suspended" state represents the simulation overhead due to the distributed synchronisation scheme.

We see from Figure 34 that simulation runtime grows gradually as the number of clients increases. The growth is due to increased time spent suspended. For fewer than 100 nodes the overhead introduced by the distributed synchronisation scheme is very low. With 200 nodes in the simulation, the overhead introduced by the distributed synchronisation scheme is approximately 17%.

Figure 35 shows the same test run with a 50ms timeslice. Again, the overhead for fewer than 100 nodes is very low. With between 100 and 200 nodes the overhead grows faster, up to approximately 65% for 200 nodes.

This overhead is introduced by processes which fall behind in virtual time This may be caused by background load on its host or GVT update messages being held up in the network. We see a lower overhead in the 100ms test because the large timeslice acts as a buffer to absorb slow GVT updates. A slow process does not block others until it is 100ms behind them. With a 50ms timeslice this buffer is half the size, so it is more common for a simulation node to fall behind enough to stall the simulation.

4.5.2 Effect of Reducing the Scheduler Timeslice

As demonstrated in Section 4.1, reducing the scheduler timeslice can improve simulation accuracy. However, this this improved accuracy comes at the expense of increased simulation overhead due to the requirement of tighter synchronisation between threads. In this section we present experimental results which quantify that overhead. All experiments in this section use 50 clients.

Figures 36 and 36 show results from the server and client nodes respectively of the connection delay experiment from Section 4.1 ran with with 50 clients. . The graph shows how the real time taken for the simulation to complete increases as we decrease the scheduler timeslice. We introduce three new time accounting states to cover blocking network operations: "connect", "read" and "accept".

We can see from Figure 36 that the increase is gradual for timeslices above 25ms, below which it increases sharply. The server spends almost all of its time waiting to accept connections. Recall that in this experiment the server is lightly loaded, so we expect it to spend most of its time waiting for connections. Progress in this test is driven by the client. We therefore turn our attention to the runtimes of the client nodes. Figure 37 shows an average time breakdown for client nodes. We see that as the timeslice is reduced the client spends a roughly constant amount of time running but the amount of real time spent suspended steadily increases, more sharply below 25ms. This is to be expected; reducing the timeslice does not change the amount of work which a simulation needs to do, reflected

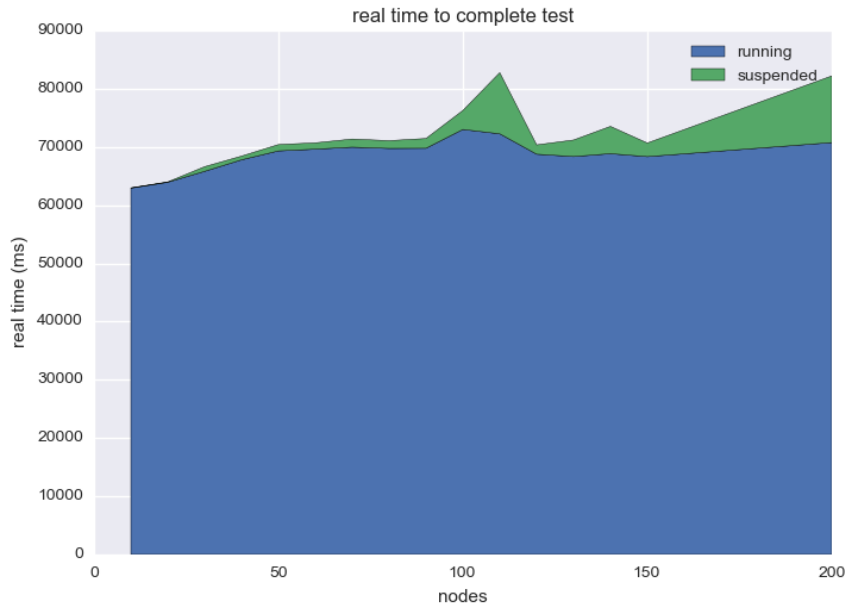

Figure 34: Real execution time for a simulation with 5 to 200 nodes with a 100ms timeslice. Averages over 6 runs.

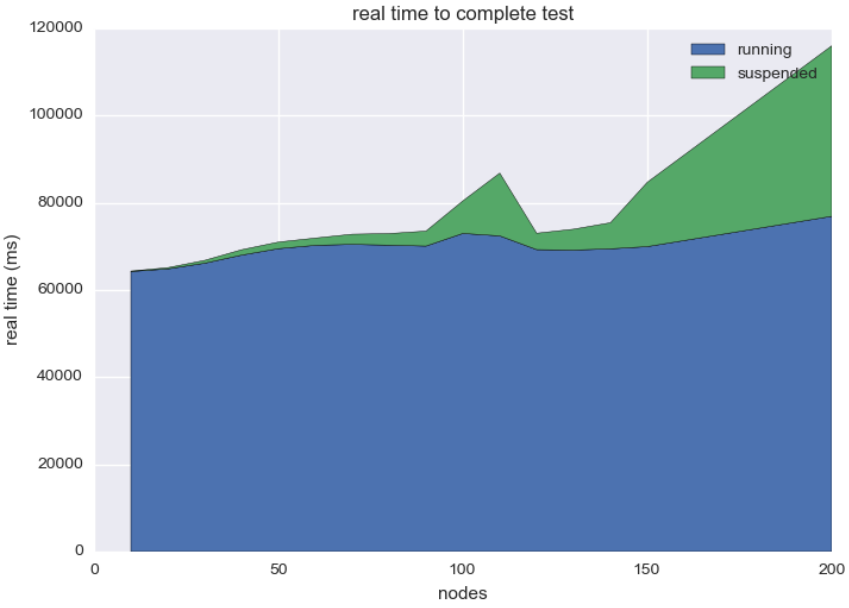

Figure 35: Real execution time for a simulation with 5 to 200 nodes with a 50ms timeslice. Averages over 6 runs.

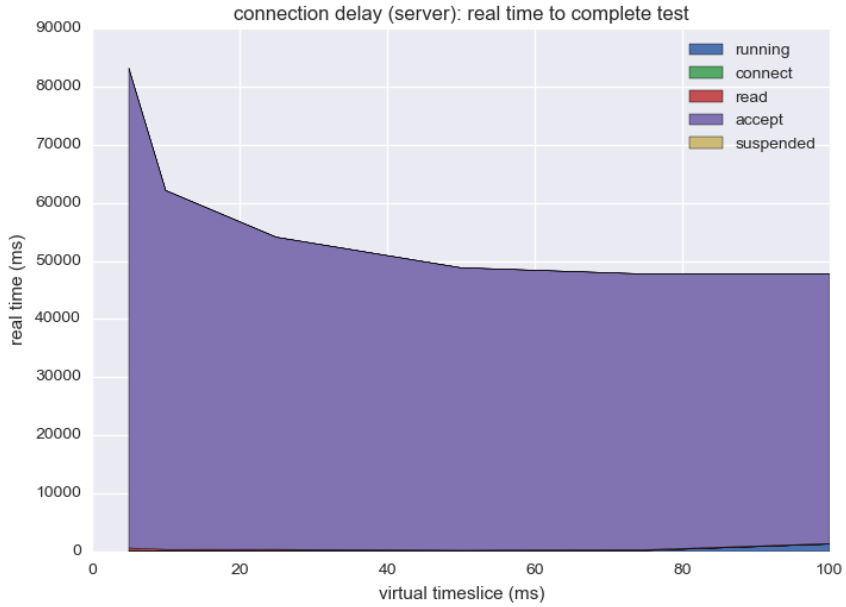

Figure 36: Breakdown of real execution time for the connection delay test from Section 4.1 on the server node. Averages over 3 runs.

in the "running" time, but does tighten the synchronisation bounds imposed on it by VTF and so increased the simulation overhead, the "suspended" time.

Similar results generated using the response time tests with a single threaded server from Section 4.2. Results for the server and client are shown in Figures 38 and 39 respectively. Recall that in this test the heavily loaded server is the bottleneck, so controls simulation runtime. This is reflected in the server now spending most of its time in the "running" state, and little waiting to accept connections. In Figure 38 we can see that simulation runtime remains stable when reducing the timeslice from 100ms to 25ms and increases sharply below 25ms. For all timeslices, amount of time the server spends running state remains roughly constant. The increasing runtime below 25ms can be attributed to increased synchronisation overhead, as evidenced by the growing amount of time spent in the "suspended" state.

A breakdown of the simulation runtime for a client node of the response time test is shown in Figure 39. Since the clients are bottlenecked by the server, we see that the client spends most of its time waiting for a response from the server ("read" time). The amount of time spent running stays approximately constant. We again observe increased simulation runtimes for timeslices less than 25ms. This is mostly attributable to a slower server (in real time) leading to increases in time spent blocked in a network read rather than because of the client being suspended by the synchronisation scheme. The server is, of course, slower because it is suspended by the synchronisation scheme.

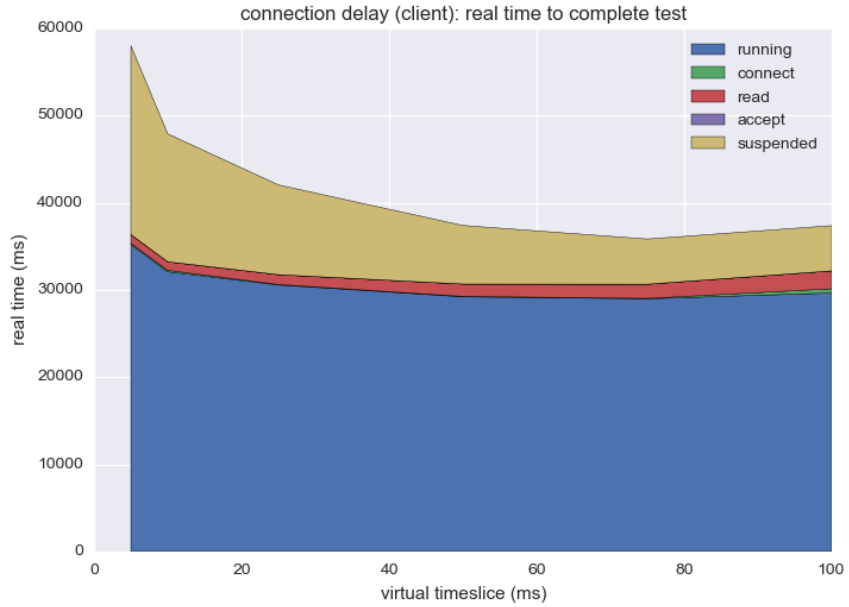

Figure 37: Breakdown of real execution time for the connection delay test from Section 4.1 on a client node. Averages over 3 runs.

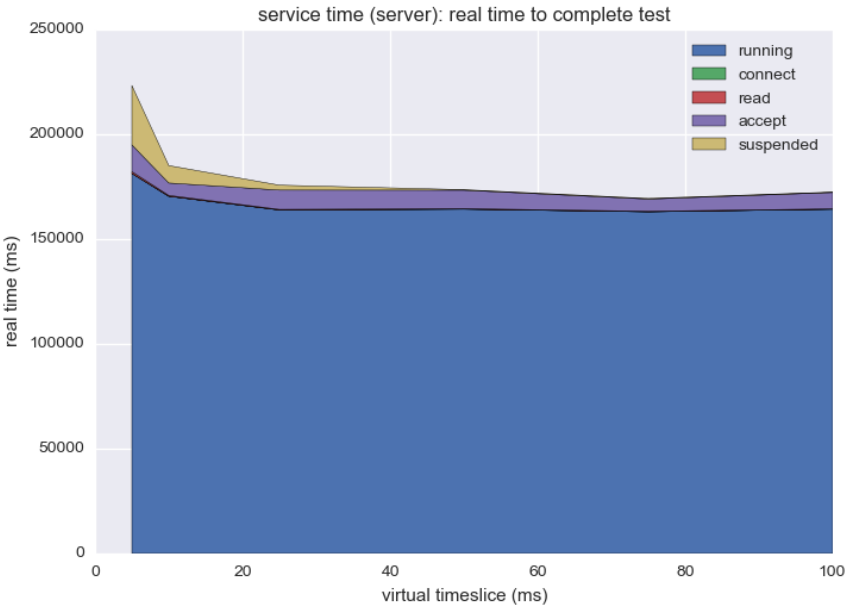

Figure 38: Breakdown of real execution time for the response time test from Section 4.2 on a server node. Averages over 3 runs.

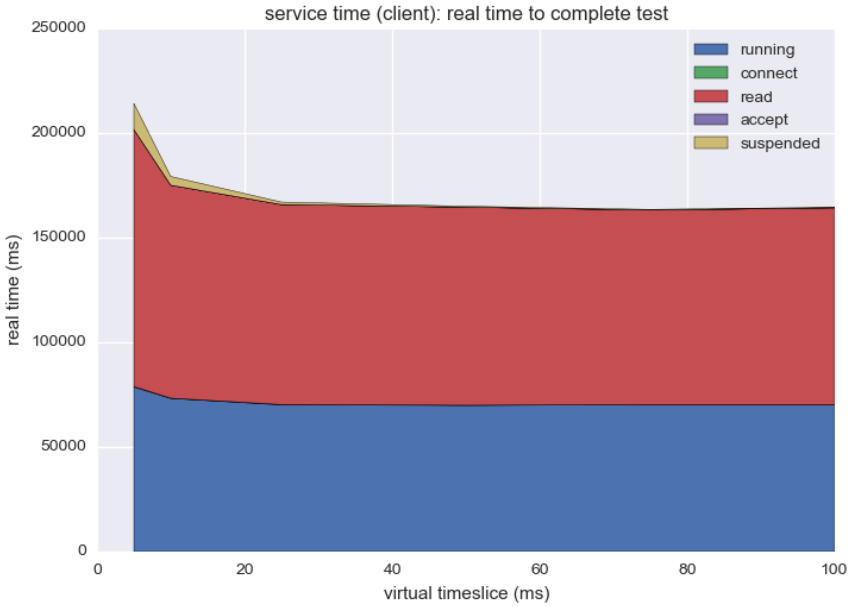

Figure 39: Breakdown of real execution time for the response time test from Section 4.2 on a client node. Averages over 3 runs.

4.6 Effect of Virtual Time on Network Throughput

We would expect to find that apparent network bandwidth is changed by VTF as messages run through the network in real time but are sent and consumed by processes at rates in virtual time, meaning that a 1Gbps link could appear to transmit data at a greater or lesser rate depending on the time scaling factor applied the sending an receiving threads. This would be similar to the time scaling effects used in network emulation.

Consider two processes which advance through virtual time at half the rate of real time communicating across a single 1Gbps link. The link will become fully saturated when a process sends 1Gb of data down the link in one second of real time. However, in that one second of real time, the process will have advanced half a second in virtual time, so the link will appear to be a 2Gbps link from the point of view of the simulated application.

However, our tests show that the situation is more complex. The results in Figure 40 are generated using a pair of programs, a "sender" and a "receiver", which are run on different machines connected by a 1Gbps link. The sender will push fixed size chunks of data into the network at a rate controlled by setting the time it waits between sending messages ("think time"). As we reduce inter-message time at the sender we expect to see a higher rate of data transfer. Comparing the transfer rates with and without VTF we find that the simulation becomes inaccurate when the think time drops below $100,000ns$.

This is due to overheads introduced by instrumentation at the receiver. Figure 41 shows

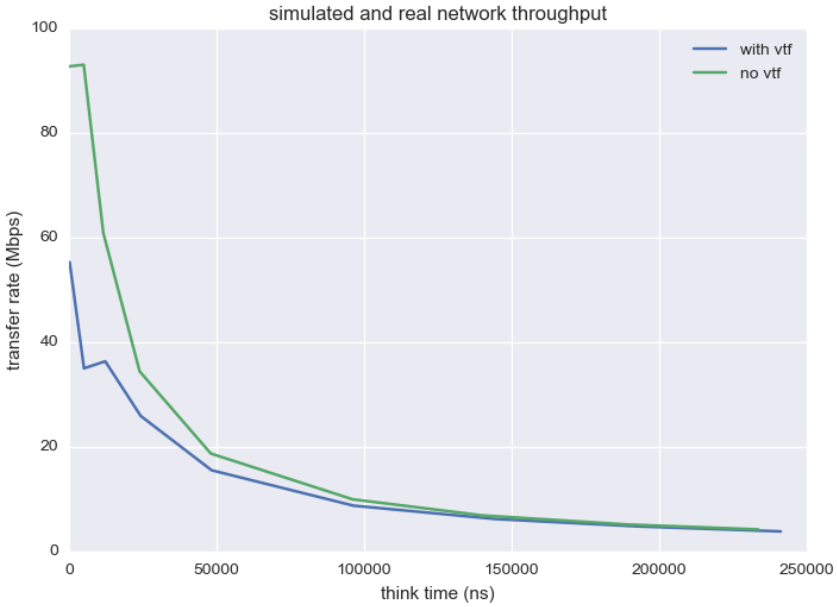

Figure 40: Simulated and actual network throughput on a single link.

Inter-message think time (ns)	Median message deliver error (ns)	Messages delivered late
250000	175006	$\sim 0\%$
100000	60567	$\sim 0.01\%$
50000	26475	3%
25000	10973	17%
12500	−31005600	87%
5000	−1336770000	91%

Figure 41: Median message delivery errors for a range of inter-message think times.

selected median message deliver errors for simulations with various inter-message think times. Notice that for larger inter-message think times delivery errors are positive, indicating a thread receiving a message which it can leap forward to and handle correctly. However, when the inter-message think time drops below $100,000ns$ a significant proportion of messages are delivered late; the receiving thread has already passed the point in virtual time at which this message should be delivered. Because of extra code executed by the instrumentation added to `read` calls, the rate at which the sender puts data into the network outstrips that at which the receiver can read it out. We therefore see a queue develop as messages collect in the network buffer leading to late message delivery.

Chapter 5

Conclusions

5.1 Achievements

We have shown our distributed synchronisation scheme to be effective. Toy examples have been used to show that VTF can now make accurate predictions about the performance of distributed systems under time scaling for CPU bound workloads to within the framework error of approxmately 8%.

We also find the approach to be scalable to medium size systems of a few hundred nodes when large timeslices are used. In particular, for a large (100ms) timeslice and small number of nodes, the additional simulation overhead is negligible.

Our experiments with *Jetty* show that VTF is making progress towards its goal of simulating real-world applications, but there is still work to do.

5.2 Limitations

We have not been able to properly evaluate our synchronisation scheme with real world systems or heavily loaded multithreaded programs due to limitations in the underlying framework. Our synchronisation scheme appears to be very sensitive to small simulation errors introduced by the framework.

Our extensions to the instrumentation framework only supports network I/O over the `java.net.Socket` and `java.net.ServerSocket` classes. While this is sufficient for toy programs, high performance applications would use the non-blocking alternatives in the `java.nio` package. It would be necessary install further instrumentation in order to profile such systems.

At this stage, VTF would not be effective at profiling applications whose performance constraints are based on network throughput because the overhead from `read` instrumentation impacts performance. We have not been able to verify our hypothesis that VTF would exhibit time-dilation like bandwidth scaling properties, but still expect it.

5.3 Future Work

This section describes a few possible "next steps" for the VTF project.

5.3.1 Virtual Time Aware End-to-End Tracing

Distributed systems are often organised in multiple tiers - for example a webserver may be backed by a number of services which themselves use databases. End-to-end tracing systems attribute activity in each tier to particular requests. Examples can be found from both industry and research. Google's Dapper [12] and Twitter's Zipkin [13] both produce Gantt chart visualisations of request activity lengths on multiple machines. A virtual time aware end-to-end tracing system would be a useful tool for visualising how potential optimisations would effect the ordering and durations events in distributed environments.

5.3.2 Evaluation using Real World Systems

A long term aim of the VTF project is to produce a performance engineering tool which can be used in the real world. In order to reach this goal, VTF needs to be tested with more real world applications to expose bugs and prediction errors that only appear in complex applications. This could be both in single process and distributed system. An interesting project would be to find a open source project in which performance optimisations have been implemented and metrics documented and see if VTF can predict the performance improvements following optimisation using the unoptimised code.

5.3.3 Code Cleanup and Testing in VTF

As more people become involved in the development of VTF the quality of the project codebase becomes more important. With many developers it is important that the project has a thorough test suite to avoid introducing regressions. Much of logic of VEX/VTF is intertwined with side-effecting code which make implementing repeatable, reliable tests difficult.

5.3.4 Investigate Prediction Errors in Multicore Simulations

The erroneous behaviour described in Section 4.3.2 which VTF exhibits in stressed multicore simulations needs attention.

5.3.5 Network Simulation in Virtual Time

As noted above, we expect VTF would exhibit time-dilation like bandwidth scaling properties, which would make it unsuitable for predicting performance of applications which

saturate the network. An application with a TSF of 3 progresses more through virtual time at a third of the rate of real time, so may push data into the network at a greater rate in virtual time that the network can handle in real time. A performance prediction based on this would be wrong, as the real application would block in `write` calls. To solve this problem VTF needs a network simulation module similar to its current disk I/O simulator. A valuable extra use VTF to predict when a change in performance characteristics would turn a CPU bound application into a network bound application.

Bibliography

[1] R. M. Fujimoto, *Parallel and Distributed Simulation Systems*. Wiley, 2000.

[2] D. R. Jefferson, "Virtual time," *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 404–425, July 1985.

[3] J. Weidendorfer, M. Kowarschik, and C. Trinitis, "A tool suite for simulation based analysis of memory access behavior," in *Computational Science - ICCS 2004* (M. Bubak, G. van Albada, P. Sloot, and J. Dongarra, eds.), vol. 3038 of *Lecture Notes in Computer Science*, pp. 440–447, Springer Berlin Heidelberg, 2004.

[4] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker, "To infinity and beyond: Time warped network emulation," in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, (New York, NY, USA), pp. 1–2, ACM, 2005.

[5] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker, "Diecast: Testing distributed systems with an accurate scale model," *ACM Trans. Comput. Syst.*, vol. 29, pp. 4:1–4:48, May 2011.

[6] N. Baltas, *Software Performance Engineering using Virtual Time Program Execution*. PhD thesis, Imperial College London.

[7] N. Baltas and T. Field, "Software performance prediction with a time scaling scheduling profiler.," pp. 107–116, IEEE, 2011.

[8] G. Charles, "A virtual time performance engineering tool," Master's thesis, Imperial College London, 2014.

[9] `http://asm.ow2.org/`.

[10] K. J. Duda and D. R. Cheriton, "Borrowed-virtual-time (bvt) scheduling: Supporting latency-sensitive threads in a general-purpose scheduler," *SIGOPS Oper. Syst. Rev.*, vol. 33, pp. 261–276, Dec. 1999.

[11] `http://kernelnewbies.org/Linux_2_6_23`.

[12] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," tech. rep., Google, Inc., 2010.

[13] `http://twitter.github.io/zipkin/`.