Profiling Distributed Virtual Environments by Tracing Causality

Sebastian Friston* University College London
Elias Griffith† University of Liverpool
David Swapp‡ University College London
Alan Marshall§ University of Liverpool
Anthony Steed¶ University College London

ABSTRACT

Real-time interactive systems such as virtual environments have high performance requirements, and profiling is a key part of the optimisation process to meet them. Traditional techniques based on metadata and static analysis have difficulty following causality in asynchronous systems. In this paper we explore a new technique for such systems. Timestamped samples of the system state are recorded at instrumentation points at runtime. These are assembled into a graph, and edges between dependent samples recovered. This approach minimises the invasiveness of the instrumentation, while retaining high accuracy. We describe how our instrumentation can be implemented natively in common environments, how its output can be processed into a graph describing causality, and how heterogeneous data sources can be incorporated into this to maximise the scope of the profiling. Across three case studies, we demonstrate the efficacy of this approach, and how it supports a variety of metrics for comprehensively bench-marking distributed virtual environments.

Keywords: profiling, benchmarking, tools, distributed, latency.

Index Terms: C.4 [Performance of Systems]—Measurement Techniques; D.2.8 [Software Engineering]: Metrics—Performance measures

1 INTRODUCTION

Effective Virtual Environments (VEs) must respond quickly and predictably, often in the order of milliseconds. To optimise effectively, developers must understand the behaviour of their VE's components. Gathering a comprehensive overview is difficult enough within single application, but modern VEs often consist of multiple processes, loosely coupled to each other and to output devices such as image generators. Systems may be distributed across machines for additional processing power or to extend their scope. Haptics systems especially emphasise this as their real-time nature often requires dedicated hardware or real-time operating systems. Profiling tools then must trace behaviours between heterogeneous runtime environments, with different levels of accessibility and transparency, while also considering the potential for partial failures, and clock-skew [3].

In this paper we propose a profiling technique designed for distributed systems. Our technique is designed to bridge the gap between low resolution, non-intrusive systems used to monitor the health of large distributed web applications (e.g. [29], [10], [30], [3]) and the highly intrusive, but precise, tools used to profile applications on a single machine (e.g. [12], [1]). In our approach, instrumentation points are placed throughout an application at design time. At runtime, these generate samples containing hashes of the system's variables. The designer chooses which variables to hash at each location. By choosing the same variables in two locations, their samples will have the same hashes at any given time, implicitly linking them. In a post-processing step all samples are collated and associations between them identified. By following a chain of associations from one sample to the next, causality can be traced through an application, and various metrics such as latency, jitter, degree of parallelism and message frequency can be extracted.

The novelty of our technique is in our use of hashing to supplant metadata or static analysis. Hash functions can be defined to be independent of the execution environment and even sampling technique - for example, they could be generated by external sensors, or recovered from video or deep packet inspection. The ability to integrate samples from heterogeneous sources allows the system to trace causality across thread, process, and even machine and layer boundaries. The precision is limited only by that of each system's clock, and their synchronization. The logical resolution is highly scalable, depending on the number of instrumentation points, which is entirely at the discretion of the developer.

In this paper we present our approach in detail. We describe how it was implemented to profile a novel distributed haptic system. In three case studies we demonstrate how its scalability facilitated quick resolution of a synchronization bug, how its comprehensiveness facilitated the comparison of three architectures, and how heterogeneous sampling techniques can be unified to characterise the latency between an internal event and its appearance in the 'real-world'.

2 PREVIOUS WORKS

Real-time Interactive Systems (RISs) have high performance requirements, for example, update rates in the KHz range for haptic feedback [25]. As such, there has long been interest in characterizing these systems for the purposes of optimization.

Traditional Profiling tools (e.g. VTune [12], YourKit [27], VisualVM [22]) provide high-resolution and can quickly reveal bottlenecks in local applications. They are not always suitable for RISs however. Rehfeld et al. [20] noted how performance gains in modern environments depend more on concurrency, which traditional profiling tools struggle with in an asynchronous context. The authors propose a set of metrics to characterise asynchronous RISs with near completeness. The metrics are defined as functions of message-passing times, which are analogous to associations in our system. Huang et al. [10] focused on predictability and semantically defined execution intervals, which are important for asynchronous RISs, but don't necessarily map to high level function calls. Their tool *VProfiler* annotates semantic intervals at design time for precise profiling of latency within a single application.

Distributed RISs introduce additional complications. Their use of heterogeneous components mean compatibility must be managed, they must be tolerant to packet loss and their higher level of concurrency leads to higher chances of design defects resulting in deadlocks and synchronization issues [3]. Classic profiling tools are typically unable to characterise middle-ware facilities like message passing [20], and rarely work across platforms [15].

*e-mail: sebastian.friston.12@ucl.ac.uk
†e-mail: e.griffith@liverpool.ac.uk
‡e-mail: d.swapp@ucl.ac.uk
§e-mail: alan.marshall@liverpool.ac.uk
¶e-mail: a.steed@ucl.ac.uk

One approach has been to model systems formally. Hanawa & Yonekura [9] modelled the error of a Distributed Virtual Environment (DVE) motion predictor and compared its accuracy to numerical simulations (∼5%) and a user study. Rehfeld and Latoschik [19] demonstrated how model checking could predict the performance of an asynchronous RIS (< 30% difference). Model checking requires that the system be accurately described in a modelling language, requiring a distinctive set of skills to those required to instrument code. However, a reliable model could reveal design defects before a system is implemented.Singh et al. [24] used profiled metrics to validate and improve on models iteratively. They captured and coded real user data in order to construct a 'user simulator' for testing things such as per-user CPU and memory load on a DVE. In a follow up study Singh & Gracanin [23] used a similar approach with more traditional metrics: execution times and call counters.

Rueda et al. [21] characterised a peer-to-peer DVE. They monitored throughput and latency, with respect to different client behaviours (message rates, and location in the virtual world). They used the response times of each client to avoid clock synchronization issues, and from repeated experiments identified the most significant behaviours affecting overall performance. Chen et al. [5] measured the sources of latency in remote rendering cloud gaming systems by using application hooks to measure the response times of salient events. Kämäräinen et al. [14] looked at remote rendering mobile Mobile Virtual Reality (MVR) systems. They comprehensively characterised the end-to-end latency using a multi-modal hardware platform including photo-sensors, touch sensors as well as the ability to time-stamp the client's network traffic. Kämäräinen et al. unified samples from these sensors with those from internal software events by synchronizing their platform's clock with that of the mobile device using a protocol similar to NTP. Casiez et al. [4] combined hardware probes and with software time-stamping in a similar manner, but relied on a low latency USB interface for their sensors, rather than clock synchronisation.

Another area with an interest in profiling distributed systems is large database-backed web services. These systems have similarities to asynchronous RISs: they are composited of multiple asynchronous nodes, running on different machines at non-trivial distances. They service thousands to millions of messages, though the response time requirements are not as tight as for RISs (<100 ms [15]). Importantly, they are often built from a mixture of black-box and white-box components.

One tool designed to profile such systems is *Iprof*, by Zhao et al. [30]. Iprof uses static analysis to associate existing log statements with top-level methods. Logs from multiple nodes are collated and these associations are used to trace causality throughout a distributed application. Zhao et al. [29] extended this idea in *Stitch*, a tool that can forgo the static analysis if log messages contain pertinent object identifiers. Zhao et al.'s [29, 30] work is most similar to ours, as they use functions of the system state to identify correspondences between samples without passing metadata. However, the temporal requirements of RISs are more demanding than Iprof or Stitch are designed for. Zhao et al. did not discuss clock synchronization. Further, Iprof had an event attribution accuracy of 88.2%, which may lead to confusion when characterising things like impulse response of a haptic system or other transient issues. Chow et al. [6] presented *UberTrace*, a tool similar to Iprof but that used a schema to which each type of log message was explicitly mapped. UberTrace also accounts for clock skew making it arguably the most (temporally) accurate tool considered so far. Lai et al. [15] introduced *milliScope*, which unifies logs from heterogeneous native profiling tools. milliScope uses metadata to link samples, tracking a request across a distributed application with perfect accuracy. milliScope is similar to our technique in its placement of unambiguously linked tracepoints, however we reduce the intrusiveness of the profiling by linking them with hashes rather than metadata.

Mace et al. [16] proposed Pivot Tracing. They introduced the *happened-before* join, allowing the construction of chains of arbitrary samples for causal tracing, as done in tools such as milliScope. It is Mace et al. that introduce the term *tracepoint*.

There are then many tools to accurately instrument applications. There are also techniques to trace causality across heterogeneous sources. However, the data sources for these approaches are often highly coupled to the system under test, or the runtime environments that they were designed for, limiting both the applicable systems and the domain of resulting profiles.

3 TECHNIQUE

We investigate a technique to profile distributed RISs such as VEs. The requirements for our profiling are that it can easily encompass new hetereogenous components - such as software written in different languages, running on different operating systems - as well as platforms such as FPGAs and microcontrollers. The profiling must also extend to the real world - sample the actual actuation of haptic arms and displays. We assume the developer can instrument their system: modify the code, or in extremis insert hooks or shims, and directly read from hardware devices or otherwise attach external sensors. We propose an instrumentation that places minimal demands on the execution environment. Developers run an instrumented system for a time to generate simple logs. These are combined a-posteriori and analysed. The logs are highly decoupled from the processing tool. We use Matlab for the analysis, but equivalent implementations could be made on a range of platforms. Alternatively, schema-based tools could even be re-purposed. In the following sections we describe simple instrumentation to generate such log files. We describe how different data sources can be unified, and how causality-tracing techniques can be applied to profile performance across thread, machine and layer boundaries.

3.1 Tracepoints

In our profiling system, an application is explicitly instrumented at design time with *tracepoints* (Table 1). Tracepoints write samples (Table 2) to a log file whenever they are encountered. Each sample is uniquely identified by the originating tracepoint (where in the system it was generated), and in time by the time-stamp. Samples do not have bounds; they represent the system state at a single point in time. An application is characterised by building a directional graph of these samples and determining the elapsed time between them. The graph is built by identifying associations between samples through hash-matching.

3.2 Hashes

Samples include hashes generated by running hash functions on a subset of the system state. Which variables to include is up to the designer. By choosing the same variables in two locations, they will compute the same hashes and so be implicitly linked. Samples have an input hash, and an output hash. This allows tracepoints to change which variables and hash functions are used to form each link in the chain, allowing causality to be traced across both technical (thread, process, machine, layer) and semantic boundaries. Example definitions and samples from Case Study 1 are shown in Tables 3 & 4.

3.3 Hash-based Control Flow

Matching is based on simple equivalence. Accurately recovering control flow requires that edges are only created between samples from truly causally related tracepoints. This can be done by ensuring all hashes are unique, or by constraining which tracepoints can match with which. In the first case, the designer would choose a sufficiently unique set of variables for each link in the chain (pair of tracepoints), so that there will be no hash collisions anywhere else in the application, or in the same location at any other time. In the

second case, the designer uses a much simpler parameterisation, but the matching process is constrained so that samples from tracepoints with potential collisions may not match. Examples of states that could be hashed are the payload for a message, the counter in a loop, or the force currently applied to a device or simulated for an object.

The parameterisation complexity is a trade-off between a-priori and a-posteriori effort. The more unique the hashes generated at design time, the more the control flow can be recovered automatically - but the harder it is to add or remove tracepoints while ensuring that the chain of matching output-to-input functions is unbroken. Alternatively, the developer specifies which tracepoints may match based on type in the analysis tool, ensuring that only the correct path between two tracepoints is recovered even if parameterisations are re-used. This provides flexibility, as tracepoints can be inserted and removed without altering adjacent hash functions - only the a-posteriori constraints. Storing associations as edges means a sample may have multiple associations in either direction, allowing control flow patterns such as branching and fan-outs to be recovered. As the samples themselves form the graph nodes (not the tracepoints that originated them), it is possible to observe how control flow changes over time.

3.4 Implementation

A drawback of many tools, even ones for distributed applications, is that they only support a subset of environments. Rather than use existing libraries with this limitation, we aim for our technique to be simple enough that it can be implemented natively.

Table 1: Tracepoint definition

Variable	Type	Description
node_name	string	Identifies program
instance	string	Identifies instance of program
tracepoint_name	string	Identifies tracepoint in program
input_hash_type	string	Indicates content type in hash
output_hash_type	string	Indicates content type in hash

Table 2: Tracepoint logged data structure

Variable	Type	Description
tracepoint	Tracepoint	Tracepoint encountered
sample_time	double	Time of data
input_hash_value	uint32[4]	Hashed data entering tracepoint
output_hash_value	uint32[4]	Hashed data leaving tracepoint

Table 3: Example Tracepoint Definition from Case Study 1

Variable	Source	Example Data
node_name	Developer Specified	”Chai”
instance	Hash of PID and MAC	4108243516105288
tracepoint_name	Developer Specified	”SceneGraphBusProcessMessage”
input_hash_type	Developer Specified	”messageid”
output_hash_type	Developer Specified	”messageid”

An implementation has two parts: the hashing functions/tracepoints in the execution paths, and a shared logging component. Tracepoints are responsible for generating the samples (Table 2) by executing the hashing functions. They then pass their samples to another component to output to the logging device (e.g. a file on disk). In our implementations, tracepoints are simple functions that pass objects to a high performance lock-free queue. A thread then gathers these tracepoints and writes them to a file. This prevents jitter in disk access times or thread synchronization primitives from interfering with the application's critical paths. Tracepoints themselves have minimal overhead, so there is little disadvantage to having multiple instances if it were necessary to, for example, hash multiple sub-states in a single location.

Table 4: Example Log Sample from Case Study 1

Variable	Source	Example Data
tracepoint	Instrumentation Call Params	(Tracepoint reference)
sample_time	Instrumentation Timing Code	15053009044.9114494
input_hash_value	Hash of Message Id	10e4b624000000000000
output_hash_value	Hash of Message Id	10e4b624000000000000

The most demanding part of this approach is the requirement of a lock-free queue. We use open-source implementations available for our environments (MSVC, CLR). The implementation approaches are well known [17], so they could be ported to other platforms, however this is non-trivial. Similarly, for hashing we use MD5. Hashes do not have to be cryptographically secure, and there are simpler fingerprinting algorithms (e.g. Rabin's Fingerprint [18]). However MD5 has many open-source implementations and is often part of frameworks such as .NET.

Implementing tracepoints is more involved than placing instrumentation points for tools like VProfiler [10], due to the need to choose which variables to hash. However, it is the hashes that allow associations to cross machine and semantic boundaries.

3.5 Numerical Precision & Epoch

Our system specifies double floating point precision for absolute time, in seconds. Many computer systems represent time as a set of discrete intervals from an epoch, where the interval resolution and the epoch vary. The lack of standardisation makes the use of SI units attractive, but the necessity of floating point storage introduces numerical precision considerations. A double can accurately represent 17 significant digits [13], therefore with an epoch of the year 1601 (Microsoft Windows systems) we could expect a resolution of 1 microsecond. We considered this sufficient. If necessary however, this could be improved by agreeing an epoch of a later date (e.g. beginning of the current year), since the only constraint is that it pre-dates all the samples in a given analysis.

3.6 Analysis

Components of the system or other instrumentation write tracepoint samples to log files. In our implementation these are simply Comma Separated Value (CSV) files with each line a concatenation of Tables 1 & 2. Log files are collated, and each sample becomes a node in a graph. Directional edges are created by matching the output hash of a node to the input hashes of other nodes. The range of destination nodes is limited. Potential matches are first filtered by the happened-before operator. Secondly, for each tracepoint, the designer specifies what other tracepoints may be immediately downstream and only edges that match one of these pairs may be created (Section 3.3). Once all possible edges have been created, processing is complete and the graph may be traversed to find causality between two samples, and the delay between them by comparing their timestamps.

This latency computation, and the other metrics described in Section 5.3, were implemented in Matlab. For performance reasons, log file parsing & graph building was performed in a .NET assembly that was loaded into Matlab. Log processing is decoupled from the logging and so can be limited to a single environment. Even so, constructing the graph is straightforward in a true object oriented language as it is mainly a procedure of filtering and matching.

4 Time Synchronization

When crossing machine boundaries, clock synchronisation becomes important to timing metrics. Due to the difficulty in obtaining a global system clock, various approaches have been explored. For their visualization tool *ShiViz*, Beschastnikh et al. [3] use vector clocks to maintain the logical time between nodes. Kämäräinen et al. [14] explicitly synchronise their clocks with an NTP-like

protocol. Rueda et al. [21] only consider relative times within each client and bypass clock synchronization considerations. Chow et al.'s [6] UberTrace system computes the clock-skew and network delay between two nodes from the message transmission and receipt timestamps at either end during RPC-like exchanges.

Our implementations rely on the system time for simplicity. This requires that the clocks of each node be synchronised. We experiment with two techniques.

In Case Study 3 we use Pulse Per Second (PPS) signals to synchronise the clock phase to that of a signal generator. PPS is a 1 Hz square wave of arbitrary width, with the rising edge in phase with the second boundary of a reference clock. We use an Amplicon PCIe GPIO card to sample the signal and detect the rising edge, then round the system clock to the nearest second. Since Windows is not a real-time operating system however, it is always possible that the algorithm is pre-empted. We can use the system's high-performance counter to detect this case, but even so it may be too late to stop adjusting the clock. As such, instead of setting the clock every second we set it once and rely on the local oscillator to maintain synchronisation after. The initial synchronisation was typically within 500 µs. This was sufficient for Case Study 3, but the clocks quickly drift ($9\,\mu s\,s^{-1}$). More advanced algorithms continually adjust the synchronisation of the local clock.

We intend our system to operate across large geographical distances, and so need a true global clock as a reference. For larger systems we use UTC and we rely on Precision Time Protocol (PTP) to synchronise individual nodes. PTP is a time-delivery protocol that can deliver time with sub-microsecond accuracy by compensating for the estimated network latency. It is common in the telecommunication and power industries, and is therefore well supported [2, 26]. As delay estimation is critical for accurate operation, devices with non-deterministic latency in the path must be PTP aware. For Case Study 2 we transmit time on a dedicated network, as shown in Figure 1. The direct connection of each computer to the time-server (Grandmaster) bypasses the non-deterministic switch and ensures a predictable latency. In informal tests, Windows 8 (and later) has been shown to maintain an accuracy of 50 µs, and a synchronisation of $\sim 200\,\mu s\,s^{-1}$ [7], which matches our observations during the experiments.

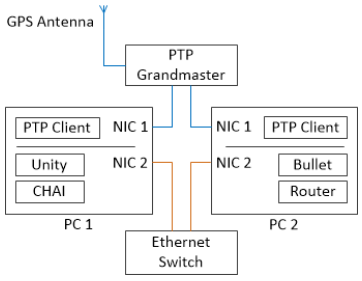

Figure 1: Example network configuration for time distribution used in Case Study 2.

5 Case Studies

5.1 Prototype System

We demonstrate the efficacy of our technique by profiling a novel distributed haptic RIS. Our system is built from up to five nodes synchronised using the scene-graph-as-a-bus technique [28]. Nodes communicated over a thin message passing library running on TCP. There is a graphics rendering node built in Unity (C#), a combined graphics rendering and physics simulation node built in Unity (C#), a physics simulation node powered by Bullet (C++), a haptic rendering node powered by CHAI3D (C++) that drove a Phantom Omni, and a simple fan-in/fan-out router (C++).

Each node is a standalone process making maximum use of established libraries and running its own UI, critical loop and networking threads. Similar design patterns are used in each. Every network connection is supported by its own thread, and all synchronization between the network and critical loop threads is based on lock-free queues. The shared scene graph is maintained through the exchange of atomic messages. As such it is highly decoupled from the message transport and there are multiple interconnection possibilities, which are described for each case study.

5.2 Case Study 1 - Thread Synchronization Debugging

In this case study we demonstrate the flexibility of hash-based tracing across multiple boundaries, by showing how our technique assisted in resolving a real bug in our networking code. This case study used the combined physics and graphics node, and Chai node to provide a minimal working example of the issue. The high-level execution and data flow of this application, along with the tracepoints, are shown in Figure 2.

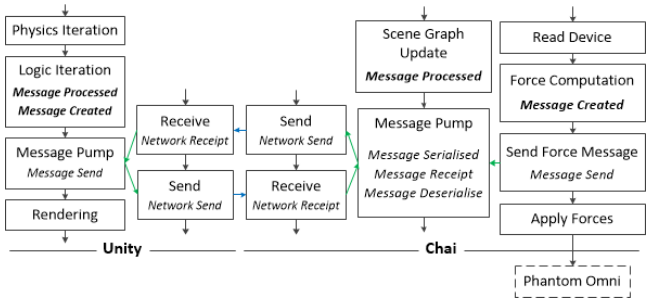

Figure 2: High-level data flow of the system in Case Study 1. Each column represents a separate thread. Tracepoint locations are shown in *italics*. The four initial tracepoints are **emphasised**. Message passing between threads is shown in green, and between processes in blue.

At runtime, we found that when touching haptics objects, they would react visually, but appear stationary kinaesthetically indicating de-synchronisation of the haptic rendering and graphical rendering scene graphs. In the ideal case, they would always be synchronised. Objects continued to accelerate so long as a force was applied with the haptic device. This indicated that latencies between nodes were asymmetrical.

Initially, four tracepoints were placed in the application hashing message ids. Recall that tracepoints represent only a single location, not a bounds, and do not necessarily correspond to individual function calls. The system was run for 12 seconds generating ~22,000 samples. These were analysed in Matlab to determine the end-to-end latency between the nodes in both directions, confirming the hypothesized asymmetry (Figure 3, Run #1).

Based on this conclusion, additional tracepoints were introduced focusing on the disproportionately latent path (Unity to Chai), and the system ran a second time (Figure 3, Run #2) generating ~60,000 samples. We hashed loop counters, message ids and message payloads, and used a-posteriori constraints (see Section 3.3) to disambiguate the paths. With this additional resolution we identified the exact link in the chain of tracepoints followed by messages between Unity and Chai that was introducing the delay. This link corresponded to a thread boundary in Unity in which the recipient was not reading all available messages at each iteration causing a backlog, and on review of this code the bug was resolved on the first attempt.

Figure 4 maps the maximum observed latency between nodes. Black indicates there is no identifiable connection between the nodes. Figure 5 shows an example of a recovered route between two nodes, as a directed network graph [11].

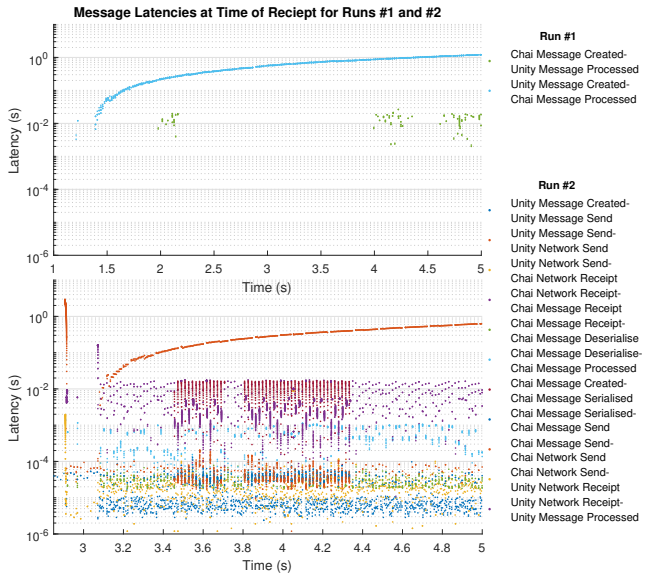

Figure 3: Plot of message passing latencies in Case Study 1 for the beginnings of two runs: #1 with four tracepoints (top), and #2 with twelve (bottom).

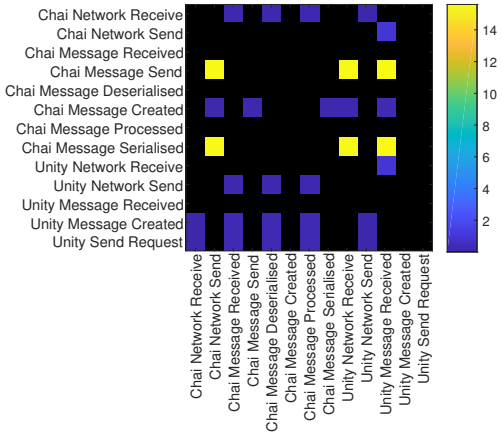

Figure 4: Table of all node permutations and the maximum observed latency (in seconds) between them, highlighting the latency issues discovered in Case Study 1.

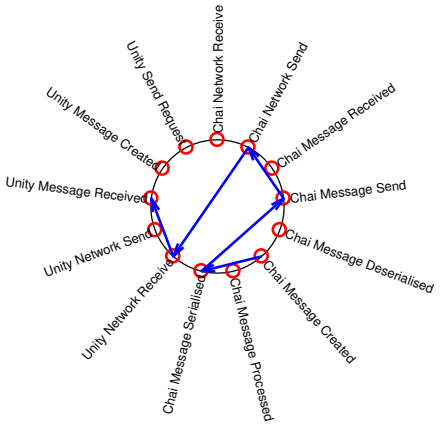

Figure 5: The route taken by a Force message from the Chai haptic rendering to the Unity physics engine.

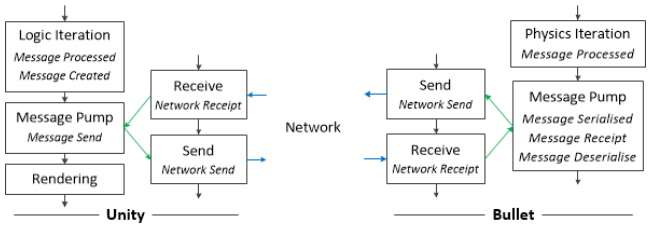

Figure 6: High-level execution flow of the rendering and physics nodes in Case Study 2. Message passing between machines is shown in blue.

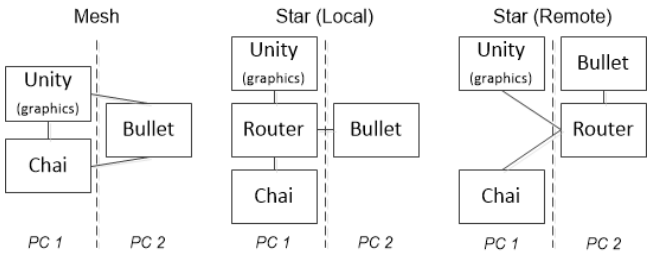

Figure 7: Three potential connection architectures tested in Case Study 2.

5.3 Case Study 2 - Virtual Network Architecture Comparison

In this case study, we profile a distributed application across multiple metrics in order to better understand the performance implications of different interconnection architectures. In this system the physics engine was placed on a separate computer, with the intent that in the future additional haptic nodes could be added and collaboratively interact with the physical scene through the scene-graph-as-a-bus scene graph synchronisation. Here we use the graphics rendering, physics simulation, haptic rendering and router nodes to build the same system with different architectures as shown in Figure 7. The Chai node had the same construction as shown in Figure 2. Equivalent diagrams of the Unity and Bullet nodes are shown in Figure 6.

To determine how this may affect performance we placed a number of tracepoints throughout the nodes, and ran the system for ~60 seconds in each configuration generating ~$4e^6$ samples. We took the same approach to hashing as in Case Study 1. The environment is shown in Figure 8. The operator behaved similarly during each capture by attempting the same task: lifting one or more items.

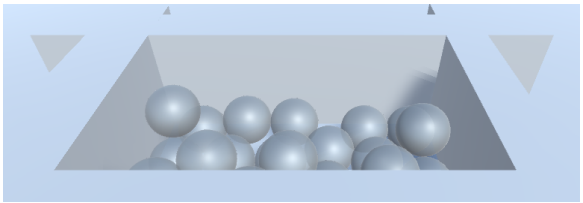

Figure 8: Simple haptics ballpit environment used in Case Study 2. The volume of the pit is roughly the same as that of the Phantom Omni.

We implemented a variety of metrics defined by Rehfeld et al. [20] that were designed for the benchmarking of message passing RISs. Their application to our system is described below; exact definitions can be found in the original publication (we have continued to use their numbering).

M3 Degree of Parallelism The number of messages concurrently 'in-flight'.

M4 Process Latency The performance of an individual node characterised by the delay between the tracepoints in the following pairs:

1. Bullet Message Receipt to Message Processed
2. Bullet Simulation to Message Transmission
3. Chai Force Computation to Message Transmission
4. Unity Message Receipt to Message Processed

M5 Latency The performance of the system characterised by the end-to-end latency in a meaningful semantic interval between the tracepoints in the following pairs:

1. Haptics Force Computation to Graphics Scene Graph Update
2. Haptics Force Computation to Physics Scene Graph Update
3. Physics Simulation to Haptics Scene Graph Update
4. Physics Simulation to Graphics Scene Graph Update

M8 Messages Per Second Average number of messages exchanged per second.

M11 Message Waiting Time Time between a message being created and beginning processing.

M12 Simulation-Overhead Ratio The time spent acting on a message compared to the time spent passing it. This is M10 in Rehfeld et al. We intend for the metric to be the same, but we do not have the concept of trigger messages in our system and cannot implement their definition exactly, so call our version M12 instead.

Measurements are shown in Table 5. Where a metric has an ideal value (e.g. latency) the best values have been emphasised. The standard deviations are high, however the dataset is also very large ($\sim 4e^6$) and an ANOVA shows that the M3-5 & M11 are significantly different between the configurations.

From our results we see that too much time is spent in message passing (proportion of end-to-end latencies (M5.1-4) expressed by M11), and that the Chai node contributes disproportionately to this (M4.3 compared to M4.1, 2 & 4). We also see that the remote router typically performs the best, with lower latencies and lower variances of those latencies, possibly because moving the message duplication to the second machine better distributes processing power. Note that this configuration has the best simulation-overhead ratio. As an absolute measure, this metric is only useful when the coverage of the system nears 100 %, however it can be a useful comparator if the coverage is consistent. One may expect the mesh configuration to have the best performance, however this is not the case. While it has higher concurrency the computational overhead of each node duplicating its messages appears to outweigh any benefits. Note how M4.1 and M4.2 change when the router process is moved remotely.

One consideration when profiling asynchronous systems is to what extent performance is dependent on user input. Ostensibly one solution would be to test under input that is in some way artificial or constrained, however the results will then only indicate the superior system under those constraints. If two systems are so close as to make user input the deciding factor this is itself a result.

One of the advantages of this type of analysis is that we can generate many measurements (36) from a single dataset, and even in noisy data detect trends that provide guidance for system design or further investigation, such as with the effect of processing power distribution M4.1, 2 & M5.1, and the significance of M4.3 for total performance.

5.4 Case Study 3 - Latency Measurement from Heterogeneous Data sources

In this case study we demonstrate how our technique can unify samples across different layers, by measuring the latency of a message between an internal instrumentation point and its appearance on a physical display in the haptic system from Case Study 1 (Figure 2).

To do so, we modify the Unity node to display a QR code containing the ID of the latest *Transform Update* message processed. The system clock is synchronised to an external PPS from a signal generator (see Section 4). The PPS also drives an LED. A high-speed (1000 fps) camera (Chronos 1.4) captures both the QR code and the LED (Figure 9). Both nodes write tracepoint logs as usual.

Figure 9: A function generator drives a PPS to a GPIO card where it is used to synchronise the phase of the system clock and an LED visible to the high speed camera. The QR code is undergoing a transition.

The system was run for ~8 seconds. After capture, a script reads the QR code of each video sample (frame of high speed video) as well as the luminance of the region containing the LED. The luminance is used to identify samples containing the rising edge of the PPS. The video is synchronised to the system clock by identifying the last QR code shown before a rising edge. Then, a corresponding sample from the software logs is found using hash matching. The time-stamp of this sample rounded up to the nearest second indicates the absolute time of the rising edge in the video. The software sample only indicates the second boundary, so any tracepoint that reliably pre-dates the QR display by $<$1 s can be used. The synchronisation of the system clock and PPS is monitored. The maximum deviation across the capture period is subtracted from the software time-stamp before it is rounded, to ensure it rounds up to the correct boundary even with imperfect synchronisation. Once the first boundary has been timestamped in this way, absolute timestamps for the remaining frames are interpolated based on the number of frames between successive PPS edges. Once timestamps have been generated for the video samples, they are integrated with the typical software logs. ~70,000 samples were generated across the 8 second run with ~8,700 of those from the video.

The highest deviation between the video and system clock was 600 µs. Due the persistence of the display, the QR code became unintelligible during switching at the beginning of each frame (see Figure 9), resulting in an overestimate of the smallest latency of approximately 6 ms - the average time the code could not be read.

The plot of the latencies between all nodes in the capture is shown in Figure 10. The dominant latencies are due to internal message passing and drawing delays in Unity, shown in Table 6 along with the total latency of the examined path. The high frequency component of the final stage (processing to display) is due to display persistence - the QR code remains visible for the entire frame, and so is matched multiple times. In some cases the code is displayed for two frames, no doubt contributing to the high average latency. This indicates our system would benefit from a low persistence display. (The occurrence of double frames and an end-to-end latency of ~60 ms was verified independently with high speed video.) These measures are far higher than is acceptable for a haptic system. Again our technique revealed a surprising source of latency in the form

Table 5: Mean and *Standard Deviation* of various metrics for the three-node distributed haptics system, in three configurations, with the best system for each metric **highlighted**

	M3 (*messages*)		M4.1 (*ms*)		M4.2 (*ms*)		M4.3 (*ms*)		M4.4 (*ms*)		M5.1 (*ms*)		M5.2 (*ms*)		M5.3 (*ms*)		M5.4 (*ms*)		M8 (*messages/s*)	M11 (*ms*)		M12
Mesh	**593**	*264*	6.8	*2.1*	3.5	*1.4*	12.4	*6.5*	16.5	*11.6*	32.6	*20.1*	**20.4**	*6.1*	19.3	*6.5*	27.5	*17.2*	137762	22.9	*12.7*	0.017
Local Router	255	*243*	**5.7**	*1.7*	**2.6**	*1.0*	15.3	*7.5*	9.4	*10.4*	35.6	*29.7*	24.2	*6.5*	**18.1**	*5.6*	26.7	*28.5*	80175	22.8	*20.9*	0.015
Remote Router	210	*80*	7.1	*3.0*	4.3	*2.1*	**11.7**	***5.9***	**8.1**	*6.9*	**26.0**	*11.1*	21.2	*5.2*	20.4	*7.2*	**19.1**	*11.8*	66811	**19.8**	*9.6*	**0.019**

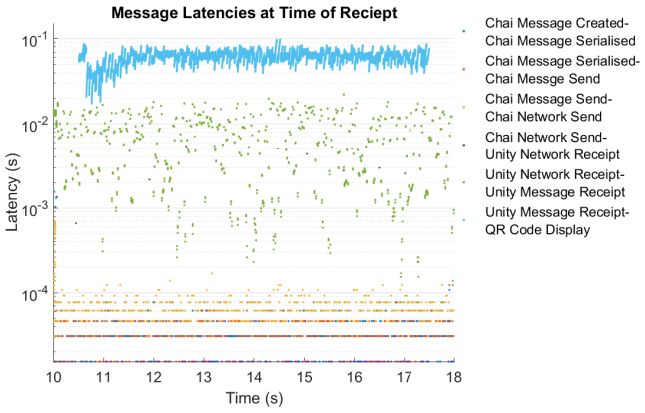

Figure 10: Plot of message latencies in the two-node distributed haptic system in Case Study 3, including the physical display.

Table 6: Message latencies of the dominant semantic intervals in Case Study 3

Path	Latency	StdDev	Min	Max
Message Created (Chai) - QR Display	0.0674	0.0111	0.0209	0.0981
Message Receipt - Processed (Unity)	0.0082	0.0107	0.0000	0.3182
Message Processed - Display (Unity)	0.0607	0.0108	0.0172	0.0976

of internal cross-thread delays, which may have been missed by less comprehensive profiles looking only at end-to-end latency, or rendering delay.

This case study demonstrates how straightforward it is to integrate non-traditional instrumentation in order to extend our profiling domain to the 'real-world'. Another variant of this would be to monitor data at various points in the network. A dedicated board could monitor packet checksums, for example, either as a 'bump-in-wire' or through port-mirroring.

6 Performance Impact

Profiling systems must have minimal impact on the system under test. While disk writes take place in a dedicated thread, hashing still takes place in the main thread. This is necessary because there are no guarantees of the lifetime of any memory being hashed, especially with black-box APIs. Table 7 shows the average performance of an MD5 hash in C++ averaged over 10^6 samples. As can be seen the hashing times are below the clock synchronisation error up to 10 kB. In practice the objects hashed are scene graph messages or local variables that never exceed a hundred or so bytes. Different hashing functions may also have different opportunities for acceleration, such as dedicated CPU instructions [8] or dedicated hardware blocks on platforms such as FPGAs, though we are not aware of any accelerators for MD5.

We used a simple log file format (CSV) in our case studies for expediency. Though it did not impact the performance of our system, it is inefficient. Each capture in Case Study 2 produced 0.5 GB of log files. Taking a Unity log as an example, 40% of each record is repeated data that could be moved into a look-up table. The remaining data could be reduced to 37% of its size by using a binary representation, or 20% in the case of tracepoints that use the same input and output hashes.

Table 7: Average execution time of an MD5 based tracepoint for different payload sizes

Payload Size	Mean Executions Per Second	Mean Execution Time
10B	1476840	0.67 µs
100B	1197810	0.80 µs
1kB	393890	2.54 µs
10kB	53178	18.80 µs
100kB	3837	260.60 µs

7 Discussion and Conclusion

Our profiling technique aims to facilitate high accuracy profiling with the ability to easily cross common boundaries in distributed RISs. A number of tools can infer data-flow entirely a-posteriori, but their efficacy is limited by the existing log formats. Other tools can achieve high accuracy, but typically do so by passing metadata or performing static analysis. This bounds the profiling to explicitly supported environments. milliScope [15] attempts to broaden these bounds by leveraging existing tools in each environment. We attempt to do so by defining a process simple enough that implementing it on a new platform is competitive with using existing tools.

The use of hashing to implicitly associate samples can introduce ambiguities about the data-flow in an application. In our experiments we used simple hash parameterisations and constrained the hash-matching to valid tracepoint pairs. This approach allows an application to be profiled at different logical resolutions, and incorporate the execution bounds of black-box APIs. It does require good knowledge of the system's architecture however. If necessary, the logical structure could be recovered automatically if each tracepoint pair used unique hash parameters. The ability to unify heterogeneous data sources is important for distributed systems, both because some components may not support modification, and also because it is necessary to instrument the real world. Kämäräinen et al. [14] and Casiez et al. [4] also unified sensor data with internal samples, but their implementations were more tightly coupled than ours.

Clock synchronization is important to distributed profiling. Like most we assume that system clocks are synchronised to a desired accuracy. UberTrace [6] has a novel clock correction scheme that does not rely on this. We do not assume that distributed applications use the traditional RPC model required by UberTrace, but integrating their approach is certainly worth investigating.

As can be seen from Mace et al. [16] and Rehfeld et al. [20], performance analysis from disparate samples is conducted predominantly by comparing timestamps. We have emphasised portability and simplicity, however at haptics rates the amount of data to be processed can become prohibitive. We had to write a native processor in C#, for speed and because graph reconstruction proved impractical without object-oriented language features. Our technique also has dependencies despite our efforts to minimise them. Lock-free queues are necessary to prevent performance degradation due to slow disk access. Hashing libraries are necessary for tracepoints to flexibly adapt to many message types and functions. We expect however that these lower level facilities will be more common than profiling subsystems that other techniques rely on.

We do assume a level of access to existing systems. Importantly we also assume a level of architectural knowledge. Where this is

the case though, our technique should be similarly easy to integrate into any type of application, including the web stacks that inspired our approach, due to the deliberately self-contained nature of the tracepoint implementation.

While our hash-matched tracepoint approach could still benefit from inbuilt clock compensation, fewer dependencies and easier automated structure recovery, it has demonstrated its potential for characterising distributed RISs. Across three case studies we demonstrate a combination of flexibility, comprehensiveness and scope that are not available in other profiling systems. All three are important for efficient optimisation of systems as complex as distributed asynchronous RISs. The instrumentation is simple enough to be implemented natively in an environment with existing hashing function and lock-free queue implementations, though we have made implementations for C++ and C# available[1]. We have also made available our processing tool and metric implementations, though in theory tools with explicit schemas such as milliScope could also be leveraged, as the principles are the same.

REFERENCES

[1] AMD. CodeXL. `https://gpuopen.com/compute-product/codexl/`, 2017. [Online; accessed 09-November-2017].

[2] K. Behrendt, K. Fodero, and S. E. Laboratories. The perfect time: An examination of time synchronization techniques. In *DistribuTECH*, 2006. doi: 10.1.1.133.4035

[3] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging distributed systems. *Communications of the ACM*, 59(8):32–37, 2016. doi: 10.1145/2909480

[4] G. Casiez, T. Pietrzak, D. Marchal, S. Poulmane, M. Falce, and N. Roussel. Characterizing latency in touch and button-equipped interactive systems. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, UIST '17, pp. 29–39. ACM, New York, NY, USA, 2017. doi: 10.1145/3126594.3126606

[5] K.-T. Chen, Y.-C. Chang, P.-H. Tseng, C.-Y. Huang, and C.-L. Lei. Measuring the latency of cloud gaming systems. In *Proceedings of the 19th ACM International Conference on Multimedia*, MM '11, pp. 1269–1272. ACM, New York, NY, USA, 2011. doi: 10.1145/2072298.2071991

[6] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pp. 217–231. USENIX Association, Berkeley, CA, USA, 2014.

[7] J. Dwight. Precision Timekeeping on Windows. *Automated Trader Magazine*, Q3(40), 2016.

[8] S. Gulley, V. Gopal, K. Yap, W. Feghali, J. Guilford, and G. Wolrich. Intel SHA Extensions. `https://software.intel.com/en-us/articles/intel-sha-extensions`, 2013. [Online; accessed 19-January-2018].

[9] D. Hanawa and T. Yonekura. Relationship between network latency and information quality in a synchronized distributed virtual environment. In *IEEE Virtual Reality 2004*, number 11, pp. 227–228. IEEE, 2004. doi: 10.1109/VR.2004.1310082

[10] J. Huang, B. Mozafari, and T. F. Wenisch. Statistical analysis of latency through semantic profiling. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pp. 64–79. ACM, New York, NY, USA, 2017. doi: 10.1145/3064176.3064179

[11] B. Huffaker, D. Plummer, D. Moore, and K. Claffy. Topology discovery by active probing. In *Proceedings 2002 Symposium on Applications and the Internet (SAINT) Workshops*, pp. 90–96, 2002. doi: 10.1109/SAINTW.2002.994558

[12] Intel. Intel VTune Amplifier. `https://software.intel.com/en-us/intel-vtune-amplifier-xe`, 2017. [Online; accessed 09-November-2017].

[13] W. Kahan. Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic, 1997.

[1] `http://wp.cs.ucl.ac.uk/sebastianfriston/publications/distributed-profiling-tools/`

[14] T. Kämäräinen, M. Siekkinen, A. Ylä-Jääski, W. Zhang, and P. Hui. Dissecting the End-to-end Latency of Interactive Mobile Video Applications. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications - HotMobile '17*, vol. 1828, pp. 61–66. ACM Press, New York, New York, USA, nov 2017. doi: 10.1145/3032970.3032985

[15] C. A. Lai, J. Kimball, T. Zhu, Q. Wang, and C. Pu. milliscope: A fine-grained monitoring framework for performance debugging of n-tier web services. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 92–102, June 2017. doi: 10.1109/ICDCS.2017.228

[16] J. Mace, R. Roelke, and R. Fonseca. Pivot Tracing: Dynamic causal monitoring for distributed systems. *Symposium on Operating Systems Principles (SOSP)*, pp. 378–393, 2015. doi: 10.1145/2815400.2815415

[17] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing - PODC '96*, number December, pp. 267–275. ACM Press, New York, New York, USA, 1996. doi: 10.1145/248052.248106

[18] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, 1981.

[19] S. Rehfeld, M. E. Latoschik, and H. Tramberend. Estimating latency and concurrency of asynchronous real-time interactive systems using model checking. In *2016 IEEE Virtual Reality (VR)*, pp. 57–66, March 2016. doi: 10.1109/VR.2016.7504688

[20] S. Rehfeld, H. Tramberend, and M. E. Latoschik. Profiling and benchmarking event- and message-passing-based asynchronous realtime interactive systems. In *Proceedings of the 20th ACM Symposium on Virtual Reality Software and Technology*, VRST '14, pp. 151–159. ACM, New York, NY, USA, 2014. doi: 10.1145/2671015.2671031

[21] S. Rueda, P. Morillo, J. Orduna, and J. Duato. On the Characterization of Peer-To-Peer Distributed Virtual Environments. In *2007 IEEE Virtual Reality Conference*, pp. 107–114. IEEE, 2007. doi: 10.1109/VR.2007.352470

[22] J. Sedlacek and H. Thomas. VisualVM All-in-One Java Troubleshooting Tool. `https://visualvm.github.io/`, 2017. [Online; accessed 09-November-2017].

[23] H. L. Singh and D. Gracanin. An approach to distributed virtual environment performance modeling: Addressing system complexity and user behavior. In *2012 IEEE Virtual Reality Workshops (VRW)*, pp. 71–72, March 2012. doi: 10.1109/VR.2012.6180887

[24] H. L. Singh, D. Gracanin, and K. Matkovic. A Load Simulation and Metrics Framework for Distributed Virtual Reality. In *2008 IEEE Virtual Reality Conference*, pp. 287–288. IEEE, 2008. doi: 10.1109/VR.2008.4480804

[25] R. M. Taylor, J. Jerald, C. Vanderknyff, J. Wendt, D. Borland, D. Marshburn, W. R. Sherman, and M. C. Whitton. Lessons about Virtual-Environment Software Systems from 20 years of VE building. *Presence*, 19(2):162–178, apr 2010.

[26] S. T. Watt, S. Achanta, H. Abubakari, E. Sagen, Z. Korkmaz, and H. Ahmed. Understanding and applying precision time protocol. In *2015 Saudi Arabia Smart Grid (SASG)*, pp. 1–7, Dec 2015. doi: 10.1109/SASG.2015.7449285

[27] YourKit. YourKit Java Profiler. `http://www.yourkit.com/features/`, 2017. [Online; accessed 09-November-2017].

[28] B. Zeleznik, L. Holden, M. Capps, H. Abrams, and T. Miller. Scene-Graph-As-Bus: Collaboration between Heterogeneous Stand-alone 3-D Graphical Applications. *Computer Graphics Forum*, 19(3):91–98, 2000. doi: 10.1111/1467-8659.00401

[29] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 603–618. USENIX Association, Berkeley, CA, USA, 2016.

[30] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'14, pp. 629–644. Broomfield, CO, USA, 2014.