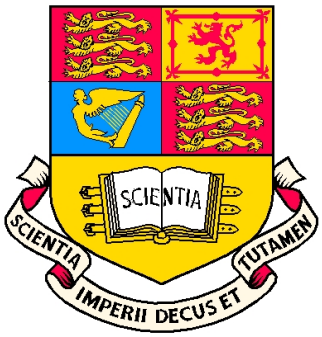
IMPERIAL COLLEGE LONDON
DEPARTMENT OF COMPUTING

Software Performance Engineering using Virtual Time Program Execution

Nikolaos Baltas



Submitted in part fullment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

To my mother Dimitra

for her endless support

Abstract

In this thesis we introduce a novel approach to software performance engineering that is based on the execution of code in virtual time. Virtual time execution models the timing-behaviour of unmodified applications by scaling observed method times or replacing them with results acquired from performance model simulation. This facilitates the investigation of "what-if" performance predictions of applications comprising an arbitrary combination of real code and performance models. The ability to analyse code and models in a single framework enables performance testing throughout the software lifecycle, without the need to to extract performance models from code. This is accomplished by forcing thread scheduling decisions to take into account the hypothetical time-scaling or model-based performance specifications of each method. The virtual time execution of I/O operations or multicore targets is also investigated.

We explore these ideas using a Virtual EXecution (VEX) framework, which provides performance predictions for multi-threaded applications. The language-independent VEX core is driven by an instrumentation layer that notifies it of thread state changes and method profiling events; it is then up to VEX to control the progress of application threads in virtual time on top of the operating system scheduler. We also describe a Java Instrumentation Environment (JINE), demonstrating the challenges involved in virtual time execution at the JVM level.

We evaluate the VEX/JINE tools by executing client-side Java benchmarks in virtual time and identifying the causes of deviations from observed real times. Our results show that VEX and JINE transparently provide predictions for the response time of unmodified applications with typically good accuracy (within 5-10%) and low simulation overheads (25-50% additional time). We conclude this thesis with a case study that shows how models and code can be integrated, thus illustrating our vision on how virtual time execution can support performance testing throughout the software lifecycle.

Acknowledgements

I would like to express my sincerest gratitude to my supervisor, Tony Field, whose knowledge, perceptiveness and positive attitude has supported, guided and inspired me throughout the course of this thesis. He has been a remarkable tutor and working with him has been an absolute privilege. I would also like to thank Paul Kelly for presenting to me the opportunity to work on this project and Alexander Wolf for bringing me in contact with the wider performance modelling community in a Dagstuhl seminar. I am grateful to my examiners Stephen Jarvis and Will Knottenbelt for their insightful comments and suggestions that have significantly improved the quality of this thesis.

I would like to particularly thank Andrew Cheadle for all the technical support and research guidance that he offered in the early stages of the project, as well as for helping us envision how our approach could be used in performance engineering. I would like to thank all the students who have been involved with the virtual time execution project: Richard Bounds, Damola Adeagbo, Andreas Matsikaris, James Greenhaulgh and Cyrus Lyons. They comprise the community that has used the virtual time execution framework and their feedback has contributed to the further improvement of the tool.

I am especially grateful to my office colleagues and friends Wolfram Wiesemann, Phoebe Vayanos, Angelos Georgiou, Raquel Fonseca and Adil Hussain for all the fun, interesting and stimulating discussions we have held over the years and for the great experience of sharing my time at the college with them.

Finally, I would like to thank my family and friends for being there for me in the better and tougher times of the research effort. I would also like to express my deepest gratitude to Delia for standing by me throughout this journey; for her constant encouragement and unconditional support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software Performance Engineering (SPE) [125] regards the activities involved towards analysing, estimating and meeting performance goals for a software development project. Performance engineers identify critical use-case scenarios from software specifications and determine, together with the management and clients, the initial performance requirements of the project. At this early stage predictions on whether these objectives can be accomplished or not, rely on performance models like Queueing Networks, Generalised Stochastic Petri Nets and Stochastic Process Algebras, properly parameterised based on the expertise of the performance engineering team. Throughout the software lifecycle measurements are used to refine the model parameters to better match the implemented code and more accurately identify discrepancies with the designated performance targets. Ideally, a well-planned and continuously performance engineered software would always reach its performance goals.

Alas, SPE is not widely used in practice: a recent survey amongst IT managers [30] shows that half of them claimed to have had performance problems with at least 20% of the applications they had deployed and attributed them to poor performance engineering strategy. Due to the significant effort and cost in model selection, design, simulation and validation, performance analysis is often delayed for after the implementation completion, thus rendering SPE a post-development exercise.

Research efforts over the last decade (as surveyed by [31], [8] and [146]) have focused on fa-

cilitating the SPE process by automatically analysing performance information and extracting performance from software engineering models (UML diagrams), appropriately extended to include non-functional attributes (like estimated response times). This approach simplifies the adoption of SPE by utilising existing architectural models, but assumes the availability of such models, a clear mapping of measurements to their components and an in-depth understanding of the application at hand.

Another common practice in performance engineering is the use of testing and profiling to identify bottlenecks in a completed system. More appropriate algorithms, caching and buffering mechanisms, code and compilation optimisations or system tuning, are then typically employed to improve the performance of critical paths, in the process of software performance optimisation. The likely effects of such optimisations on the overall performance of the software rely on the expertise of performance engineers to analyse the software behaviour, diagnose the causes of bottlenecks and suggest solutions.

Despite the many significant advances in performance modelling techniques and tools the model-based SPE methodology is currently not part of mainstream software engineering. This is in contrast to performance profiling which represents the de facto method for understanding the performance behaviour of a system, but at the post-development stage. The objective of this thesis is to develop a novel approach for lightweight integration of performance models and profiling measurements *throughout the software lifecycle*. Crucially, our approach facilitates the application of performance testing techniques on partially-complete software and constitutes a first step towards a new performance engineering methodology, where models and code can be used interchangeably.

1.1 The idea

The main idea that underpins this thesis is to simulate the execution of partially-implemented or completed programs in *virtual time*. Under the auspices of *execution-driven simulation* [32], our approach determines the progress of a program's execution, whilst profiling its performance

in virtual time. Virtual time is accounted for either as a measurement of the native execution of code on the simulation host (simulating the normal execution of the program), or as a transformation of that duration (e.g. simulating time-scaling optimisation effects) or as the time predicted by a performance model. The total virtual time of the program is the performance prediction for its duration, whilst the behaviour of the application is linked to its progress in virtual time. Using dynamic instrumentation techniques, our approach is able to offer both profiling and optimisation capabilities at the method-level. No changes to the source code of the simulated program are needed, nor does the approach require the extraction of models from code or transformations of the program source code. However, we do propose an empirical prediction scheme that compensates for I/O operations and DMA behaviour.

A key objective is to avoid the need for low-level system models, such as CPU [6, 149], memory [26, 118] or I/O [2, 101] models. We are thus concerned with the understanding of the application's performance on a given host platform, essentially using the available code itself as a "model". Since one of our key contributions is a system for integrating code and models it would be straightforward in principle to integrate models of the underlying (host) system. This, however, is a whole different exercise that lies beyond the scope of this thesis.

The objective of this research is to frame our work on the SPE map, present the virtual time execution idea, create a prototype simulator, investigate the trade-offs between accuracy and overhead and evaluate the potential for future research based on this approach.

Our thesis is that the virtual execution paradigm can be used as a new software performance engineering approach, that integrates performance models and code in order to provide performance predictions of typically good accuracy and low overhead via high-level execution-driven simulation throughout the software lifecycle.

1.2 Contributions

In this thesis we explore the feasibility of the virtual time execution idea for software performance engineering. We introduce this methodology using a bottom-up approach: from the

scheduling and profiling aspects of virtual time execution, to the application interface and the concepts of defining and including performance models. Each chapter presents the corresponding prototype systems, elaborating on the implementation details that highlight the features and challenges of virtual time execution. Our specific contributions are as follows:

- We introduce the virtual time program execution methodology and its prototype in a language-independent Virtual time Execution (VEX) core that enforces a round-robin schedule on the threads of an existing application, as if they were executed under user-defined time-scaling specifications at the method-level. This is presented in Chapter 3.

- We describe a methodology to include I/O in virtual time, by estimating I/O durations to reproduce thread-parallelism due to DMA in virtual time. This is elaborated in Chapter 4 together with the investigation of empirically-based runtime prediction approaches for I/O durations in the VEX prototype.

- We present a system for interfacing VEX's functionalities to unmodified Java applications, called Java Instrumentation Environment (JINE). The design and implementation of the JINE and the Java-related problems encountered are elaborated in Chapter 5.

- We verify and validate VEX and JINE as measured by their ability to predict real execution times of unmodified small and medium size Java benchmarks; this is presented in Chapter 6

- We describe an extension to VEX that simulates multicore architectures, by synchronising the timelines of the virtual cores and its validation on synthetic and real benchmarks (Chapter 7)

- We propose a SPE methodology and system prototype that offers continuous performance testing throughout the lifecycle by integrating executable code and performance models (Chapter 8).

1.3 Publications and Statement of Originality

I declare that this thesis was composed by myself, and that the work that it presents is my own, except where otherwise stated.

The following publications arose from work conducted during the course of this PhD:

- Modelling Analysis and Simulation of Computer and Telecommunication Systems 2011 (MASCOTS) [10] discusses the idea of executing programs in virtual time as a means to investigate the overall effect of optimising specific methods. The methodology and part of the results presented in Chapters 3 to 6 are based on this paper.
- Quantitative Evaluation of Systems 2012 (QEST) [11] presents our approach for integrating performance models and code in virtual time. Chapter 8 is largely based on this paper.

Chapter 2

Background

The virtual time execution approach lies at the intersection of the Software Performance Engineering (SPE) and the simulation research areas. This chapter discusses related work in these fields. As our virtual time execution requires an I/O prediction module, we present work on the black-box modelling of the I/O subsystem. A brief overview of Java, on which our virtual time implementation focuses, is also provided.

2.1 Software Performance Engineering (SPE)

Performance is the result of the interaction of system behaviour with resources ([146]), where by resources we refer to CPU, memory, network bandwidth, buffers and locks, threads and processes. The limited number of such resources results in the contention for their usage, which in turns leads to performance problems (*bottlenecks*). These can be quantified in terms of various metrics like response time, throughput or resource utilisation.

Performance analysis deals with two aspects of a system: *responsiveness* and *scalability*. The response time and the throughput constitute the main points of interest for the former, while their variations as the system load increases determine the latter. Moreover, since resource consumption affects system responsiveness and scalability, finding the optimal solution for each

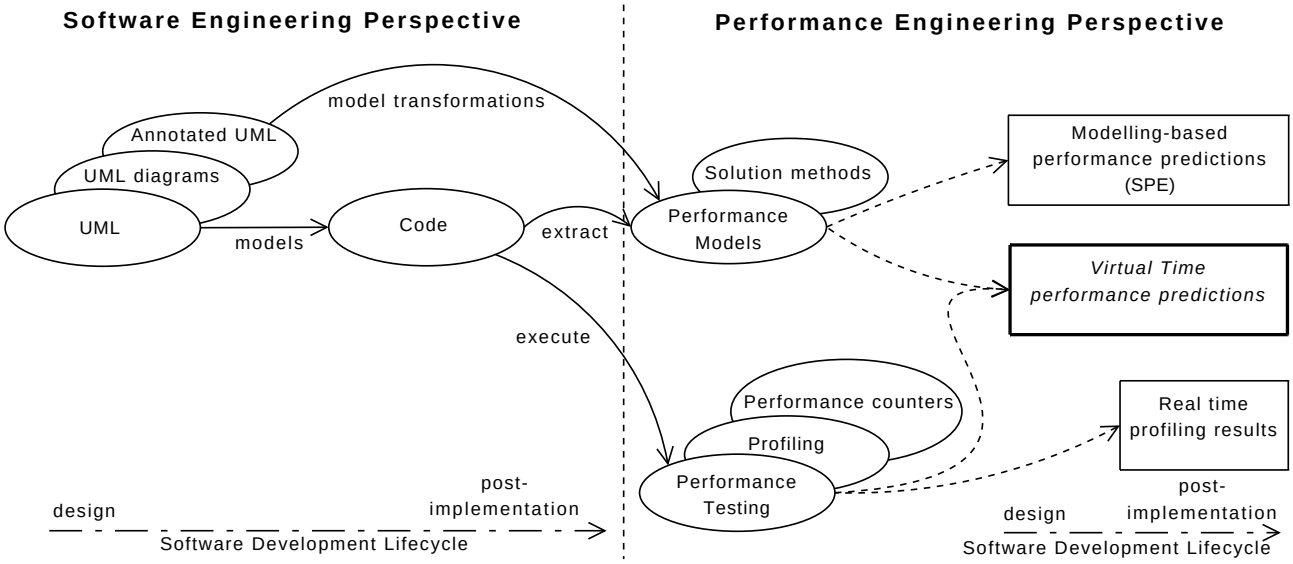


Figure 2.1: The role of the Virtual Time execution in Software Performance Engineering.

system in order to satisfy the designated performance requirements, comprises a challenging task for performance engineers.

The field of software performance engineering (SPE) is based on the pioneering work of Smith and Williams [125]. SPE is a systematic, quantitative approach to constructing software systems that meet performance objectives. It uses predictions to evaluate trade-offs in software functions, hardware size, quality of results and resource requirements. The modelling formalisms presented in this chapter are extensively used in this procedure. These models are adapted over time to better match our knowledge of the system under study, by using measurements on implemented parts (*profiling*). To provide performance predictions, SPE uses the model solution methods presented in Section 2.1.1. In principle, SPE can make use of models that target the lower-level implementation details (e.g. caching effects, I/O requests) of the software under test on a particular platform. However, this would require detailed knowledge of the system to build such models and low-level system events profiling to populate their parameters.

The description of SPE highlights the importance of two central notions in the process: *modelling* and *profiling*. These techniques are used in conjunction and in different stages of software development to ensure that the end product meets its initial performance requirements throughout the cycle. We demonstrate how the "Virtual Time performance prediction" methodology of this thesis relates to these notions in Figure 2.1: performance models and performance testing

of available code are combined under a virtual execution platform to estimate the performance of the end product throughout the lifecycle. Early stages mainly simulate models and run code to a lesser extent, while latter ones primarily performance test code and only simulate models of partially completed components.

To clarify this relation, we provide an overview of the fundamentals of performance modelling and profiling. Specifically, in Section 2.1.1 we describe how "Performance Models" and their "Solution methods" have been traditionally used to describe and analyse the performance of systems and how such models can be extracted from "Annotated UML" diagrams as a means to automatise the process. In Section 2.1.2, we describe measurements-related concepts in a top-bottom fashion: from the "Performance Testing" process, to "Profiling" mechanisms used for it and "Performance counters" used in profiling.

2.1.1 Model-based SPE

During the initial stages of software development, architectural and design choices need to be made. Performance models are used in this case to describe how the operations of each architecture affect the consumption of available resources and thus the final performance of the system. This approach might lead to early detection of performance problems that could otherwise require extensive re-engineering at a later stage. Performance models are used in different stages of the software lifecycle as means of evaluating how changes in existing systems, like different implementations or additional resources, affect responsiveness, how well an existing system scales-up in dealing with a particularly high workload etc.

Performance models

Performance models are based on some well-known and widely researched formalisms, which accommodate their construction and analysis.

Queueing networks are graphical models that describe systems in terms of waiting queues and servers linked in a network. For computer systems, this relation corresponds to the number of

processes waiting to be served by a resource. A queueing network is parameterised to denote the different characteristics of each system, like the process arrival rate, probability distribution of the service time in each server and size of each queue (if limited).

Answers to questions regarding service utilisation, average response time and system throughput are amongst the most useful ones that can be provided by the analysis of a queueing network. Overall, queueing networks provide a good balance between accuracy and cost [78].

There are a number of extensions to the basic queueing networks like Layered Queueing Networks (LQN) [116] or Extended Queueing Networks (EQN) [78] targeted respectively at layered server architectures and systems with simultaneous resource possession.

Another graphical-oriented modelling paradigm is provided by Petri nets [88]. These are directed bipartite graphs, comprising of two different types of nodes: *places* and *transitions*. States are represented by *tokens* located in places. These determine the dynamics of the system: whenever all places with arcs to a transition have at least one token, then the transition *fires*, thus removing a token from each of these places and forwarding a token to the places to which this transition has arcs.

Petri nets are particularly useful for modelling systems where issues like synchronisation and mutual exclusion are of primary interest, like distributed system or communication protocols. A stochastic extension (Generalised Stochastic Petri Nets) associates *firing time* with each transition. When a transition is enabled, i.e. each place with arcs to this transition has at least one token, then a delay period is determined according to the firing time. Once this expires and if these places still have at least one token, the transition fires.

Process Algebras [59] have been introduced into the field of system analysis as a formal (algebraic) way to model the behaviour of complex computer and communication systems. In performance analysis, Stochastic Process Algebras (SPA) have been used in order to provide quantified information for times in terms of random variables, for example as numbers specifying delays when various actions occur. In Markovian Process Algebras (MPAs) the firing times are negative exponentially distributed random variables. The underlying transition system is

then a Markov process which leads directly to numerical (and sometimes) analytical solutions. The Performance Evaluation Process Algebra (PEPA) [60] is a leading example of an MPA, featuring characteristics like composition of many smaller models in order to describe larger and more complicated systems.

Solution methods

Performance engineering involves quantifying system performance parameters, like response time, resource utilisation, queueing lengths etc, typically at equilibrium.

Analytical methods can be used to acquire such results, but only for a restricted class of models, like Markovian queueing networks. In such cases, computationally simple equations are used to evaluate the system's performance as a function of its performance parameters. Different values can then be used to identify the effects that a parameter change will have on the system.

Numerical methods are used for classes of models for which the equations that describe them do not possess an analytical solution. Depending on the stochastic information present in the high-level system description, various types of system state equations which mimic the dynamics of the modelled system can be derived and solved by appropriate algorithms. For example, a detailed description on the application of such typical linear algebra methods for Markov Chains can be found in [17]. However, numerical methods may suffer from state-space explosion, while having a higher computational complexity than analytical methods.

Simulation [76] is the most general way to analyse a performance model, covering systems whose performance is described by stochastic processes to ones where it is described by UML profiles [9]. In a Discrete-Event Simulation (DES) events occur at discrete points in Virtual Time, updating at each of them the global state of the system. Distribution sampling and random-number generators are used to simulate the model time-delays and the possible variability in the execution patterns of the simulator *sample paths*. Since different sample paths may provide different results, statistical analysis of the simulation output is required. Tools like JMT [16] and JINQS [45] use DES approaches to solve performance models.

Extraction of models from Annotated UML and code

The Unified Modelling Language (UML) is traditionally used for system modelling and has found wide applicability in the software engineering world. Since its inception and proposal by OMG, UML has seen significant changes to adapt to the needs of modern software modelling. Hence, a variety of diagram types (class, sequence, deployment, use case etc) is supported to better match modelling needs for different notions in various stages of the software development.

Annotated UML profiles, namely packages that extend the UML metamodel in a constrained way, have comprised the major approach to addressing performance issues. In this respect, OMG has proposed three profiles for UML to incorporate descriptions for non-functional requirements (NFRs): the UML Profile for Schedulability, Performance and Time (SPT) [103], the UML Profile for Quality of Service and for Fault Tolerance (QoS) [105] and more recently the UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [104, 85]. Overall, UML SPT has been widely used, though its simplicity has come to the cost of its customization and flexibility capabilities. UML QoS has a broader scope [15], supporting notions like Fault Tolerance and Dependability. Following critical aspects on these profiles [147], the OMG proposed the most recent UML MARTE profile. MARTE allows users to define their own non-functional-properties (NFP) types, qualifiers and measurement units, thus lifting the inherent limitations of the predefined descriptions of SPT. Among others, this approach overcomes the problem of lack of semantic description for the expression of time intervals between arbitrary events. Finally, the UML MARTE profile enables the logical and physical view and description of hardware components, denoting their performance or requirements specifications like throughput, power consumption etc.

Although the UML profiles enable software engineers to denote expectations of NFPs, they do not directly provide performance predictions. Smith and Williams [125] propose the identification of performance-critical use-case scenarios in UML diagrams. These guide the construction of performance models called *Execution Graphs* (EG), which are then transformed to queueing networks and are solved via one of the methods described in Section 2.1.1. The field has acquired significant research interest, leading to a variety of techniques to transform different

types of annotated UML diagrams to queueing networks, stochastic process algebras or simulation models. Such methods are surveyed by Balsamo et al. in [8], by Woodside et al. in [146] and more recently in [31].

Performance models can also be extracted directly from code execution. Previous work on this involves building models like Layered Queueing Networks (LQN) from execution traces of specific scenarios in [61, 66, 96] or PEPA models from multiple refinement passes of annotated C code [126]. As the idea that underpins this thesis regards the *co-existence* of performance models and code rather than the extraction of the former from the latter, such techniques are only relevant to the context of our work.

2.1.2 Measurements

Performing measurements is the most accurate way of characterising the performance behaviour of a software system, as they take into account all performance contributing factors from the application-level, to the operating system and system-architecture level. Following the classification found in [97], the main metrics of interest can be classified to "service-oriented" ones, like *response times* and *availability*, and "efficiency-oriented" ones like *throughput* and *utilisation*. In this section we present the measurement-based techniques that are used in the latter stages of the software lifecycle in a top-bottom fashion: from the high performance testing level to the lower-level profiling approaches and performance counter issues.

Performance testing

The experiments that take place post-implementation on a (preferably) stable codebase to determine whether the developed software system fulfills its predefined performance requirements are called *performance testing*. Identifying bottlenecks and investigating software *optimisations* or system parameter *tuning* to overcome them, constitutes another significant part of the process. Contrary to functional testing, that targets on the widest possible code coverage, performance testing addresses only a few predefined performance-critical scenarios, that typically

relate to a Service Level Agreement (SLA) for the system under test.

To acquire meaningful results during performance testing, the usage scenarios are tested via *real* or *synthetic* workloads. Real workloads probe a typical usage of the system, but are less controllable than synthetic ones. On the other hand, *characterising* and generating a usage representative synthetic workload is performed through special-purpose scripts or commercial load generation tools. Depending on whether the workloads simulate the behaviour of the system under normal or extreme workloads, we distinguish between *load* and *stress* tests respectively [91]. Analysis of the results in each case leads to conclusions, regarding performance bottlenecks and optimisation possibilities. Topics like the types and selection of workloads and performance tests or statistical analysis techniques exceed the direct scope of our research, but are covered extensively in existing literature [67, 80, 81, 97].

Performance testing typically prerequisites a complete and stable codebase. To the best of our knowledge the only published work which, similarly to this thesis, relates to performance testing of incomplete software, is proposed by Denaro et al. in [36]. In this work performance testing on middleware-based software is applied early in its development lifecycle. The solution replaces unimplemented methods with stubs [46], whose expected performance is disregarded, under the assumption that the main performance results are based on the deployment of the off-the-shelf implemented components. Although this assumption applies for commercial off-the-shelf (COTS) based software, it cannot be possibly generalised. No models are used in relation to the stubs and no functional or non-functional requirements are enforced on them, besides calling the methods required and having valid input and return values.

Profiling

The process of extracting measurements that quantitatively describe the performance behaviour of a software system is called *profiling*. Such metrics are execution times, instruction or clock cycle counts, system events like cache misses or page faults, memory usage statistics, bytes read and written to an I/O device (disk or network) etc. Matching measurements of these parameters to specific software processes, methods or actions yield the desired response times,

utilisation and throughput metrics. *Call-graph profilers* are extensively used to report program execution patterns. To allow for per method or per stack-trace aggregation of event or time counters, we distinguish between *flat* method and *path* profiles. Reported method profiles can be *inclusive* or *exclusive*, depending on whether method counters include their submethods or not.

Profiling is done via *sampling* or *instrumentation* on the application- or the system-level. Sampling profilers poll the values of system timers, resources and event counters at regular time intervals. A program profile is a histogram of the execution of a program: it describes the progress of the performance metrics in time or reports statistics for the usage of a resource based on the samples gathered. Linux system tools like *top*, *oprofile* and *vmstat* are sampling profilers, reporting CPU utilisation, processor events and virtual memory related parameters respectively. The Java *hprof* tool is a sampling application-level profiler, that acquires samples in millisecond-level intervals and reports the percentage of a method execution time to the total time, based on the number of stack-trace samples that included that method. Although sampling is lightweight and widely applicable, it can be inaccurate in the case of events of interest being lost between consecutive samples,

Instrumentation regards the addition of code that is only used to measure the system's performance. This code can be built directly into the profiled application to time events or into the host-operating system to monitor parameters like device or CPU utilisation. To account for the difference in the resource of interest (e.g. time), instruments are usually inserted before and after a method invocation. Event instruments can be added anywhere to denote the occurrence of an event. Instrumentation code may be inserted manually on a limited-scale or if this is supported by a special purpose API, like with the Application Response Manager (ARM) [71]. Typically, however, instruments are added automatically by using some form of tool. For example, the IBM Rational Quantify [64], or the aspect-oriented programming tools proposed by Pearce et al. in [110] for Java, automatically insert the profiling code into the source code of the profiled applications. An alternative way is to add instruments at compile-time. This happens with the *gprof* profiler, where timing and counting instruments are added by the *gcc* compiler, when a specific compilation flag is selected. Statistical sampling is then used to poll

these values at runtime. Other profilers like the commercial Intel VTune Amplifier XE [65] or JProfiler [42] let the user decide whether the profiling will use sampling or instrumentation. More recent approaches to instrumentation suggest the addition of instruments during runtime, with tools like ATOM [44], Pin [111] or Dyninst [144] (the latter used in the popular Paradyn tools project [94]). ASM [23] and BCEL [33] are dynamic bytecode instrumentation libraries for the Java Virtual Machine (JVM).

Exhaustively instrumenting all methods or events can lead to a significant performance penalty to the execution time of the profiled application. It is also possible that this overhead *perturbs* the reported results, by changing the program's execution behaviour. Previous work shows that, due to such *observer effects* [83], different profilers may end up reporting different performance hot-spots [100]. Malony et al. [87] measure the overhead per instrument and compensate for it in the reported results. They conclude that though in many cases this is useful, differences at the nanosecond-level on the overhead per instrument measurements, may significantly affect the success of the compensation. Besides, the method does not deal with overheads in thread synchronisation times: overheads from instruments executed by threads running in parallel with waiting thread A, are still reflected in the total waiting time of A. As the problem exacerbates, when short methods are executed at a high-rate, a selective [37] or adaptive [41, 94] instrumentation technique can be used to avoid deploying or removing instruments from such methods.

Performance counters

On the lowest-level of the measurement process we have system timers and hardware performance counters. They are exposed to the user-level via system library functions, like the Linux functions found in <sys/time.h> or through specific APIs, like the Performance Application Programming Interface (PAPI) [108] used in this thesis. PAPI has been introduced as a consistent high-level interface for usage of hardware performance counters that can be found in most modern microprocessors. In Linux, PAPI builds upon the *libpfm* and *libperfctr* user-level libraries, which use the *perfmon* and *perfctr* kernel modules.

System timers and event counters are updated, according to the frequency that the system internal timer generates interrupts to schedule the next process [24]. As such interrupts occur at the ms-level and many low-level system events at the ns-level, the system counters offer an approximate value (missing events since the last context switch), which is nevertheless adequate for large-scale performance tests. Apart from time, system counters usually measure operating system events, like signals, memory allocation, page faults, bytes transferred to and from devices etc.

Lower-level performance tests need a finer-grained time measurement and might be more interested in processor events, which are offered by the hardware performance counters. Different processors may have different event counters, though they all have a *time stamp counter (TSC)* [20], which is updated at every clock cycle. Event counters typically relate to instruction counts, cache and TLB events and are stored in CPU registers.

Both system and hardware performance counters distinguish three types of time. *Elapsed-* or *real-* time is tantamount to wallclock time. *User-* or *CPU-* time is the time that a process or thread is executing in user mode. The CPU-time measurement disregards the execution times of processes that share the CPU with the process of interest in a multitasking environment. The CPU-time counter is also left unmodified when the thread is executing in kernel mode: executing a system call, being context-switched or blocked, queueing for a resource or waiting for I/O. The *system time* counter of the process can be used to acquire such times. The *time* system call exemplifies the difference between the three on a Unix system.

An important aspect of measurements is their non-determinism, i.e. the fact that successive measurements can yield different results. Although the same CPU instructions are always processed in the same way, the multitasking environments of modern operating systems lead to processes being context-switched, so that others can make progress. By doing so, they may increase the load on shared resources (e.g. disks), whilst altering the caching, memory access and branch predictions patterns [24]. Clearly, different processes have different such behaviours. Hardware interrupts and asynchronous I/O interrupts affect the cycle count of the executing processes [141, 133]. Mytkowicz et al. [99] have found that even the size of environmental

variables or the order in the executable linking, may change the measured results, as they alter the memory alignment of the program. In [150], Zaparanuks et al. identify factors that affect the measurements of hardware performance counters, like the selection of API for access to hardware performance counters, the number of enabled counters and the duration of the benchmarks. Furthermore, Java applications are also susceptible to additional non-determinism, due to the timer-based method sampling that affects JIT compilation, asynchronous garbage collection, code optimisations and memory access pattern [51]. For all these reasons, measurements are repeated and statistical methods are employed to acquire meaningful results [67, 80, 119, 51].

2.2 Simulation

Computer simulation offers accurate and flexible analysis techniques for complex systems [67]. Computer system simulators are programs that model the inner workings of a system, updating its state upon the arrival of new events. The objective of computer system simulation is to support system design, by exploring different hardware implementations or performance tuning, by investigating the performance behaviour of future (*target*) computer architectures under various workloads. As simulators are expandable, easier to work with and less expensive than hardware implementations, they provide an excellent testbed for computer architects and system designers. Typical trade-offs of simulation concern level of detail against development time and simulation execution time versus prediction accuracy.

The virtual time execution relates to simulation in various ways:

- It borrows ideas from direct execution execution-driven simulation.
- It has similar objectives to real time simulation and network emulation.
- It uses the notion of Virtual Time presented in previous work in a new context.

This section presents related work in each of these research areas.

2.2.1 Execution-driven simulation

Event generation

The simulated workloads define the sequence of events that are processed by the computer system simulator. We can identify two types of simulation in terms of event generation.

In *trace-driven* simulation [134], the workload is used to generate a trace of its execution (i.e. the functional behaviour of the system), including instruction types, memory addresses accessed, the sequence of disk blocks requested etc. Trace-driven simulation has a high requirement in terms of memory and may suffer from inaccuracies for multiprocessor workloads [52], as it does not reflect the effects of timing-dependent issues on the target host on the trace.

These problems are overcome in *execution-driven* simulation [32], which interleaves the functional and timing behaviour of the system. A program that should be able to run on the target host is executed unmodified on the simulator. The simulator traps events (like instruction fetches, cache misses, memory accesses etc) and controls the functional behaviour and time progress of the simulation. The selection of such events and the actions taken upon their occurrence determine the tradeoff between accuracy and speed. We review existing work on execution-driven simulation in the following sections.

Cycle-accurate simulation

Execution-driven simulators reproduce the behaviour of a target system, by providing the same output under the same workload on a simulation host (assuming deterministic code). A simulator can provide functional consistency with the target system, by interpreting each instruction of the designated workload to precisely update the state of a model of the target system and the simulation time. This solution, called *cycle accurate simulation CAS*, models the microarchitecture of the target system at the CPU-level, allowing simulation of features like pipelining, execution re-ordering, branch prediction etc. This provides high flexibility and accuracy, but at the cost of low simulation speed.

SimpleScalar [6] is a mature simulation tool, supporting various architectures (Alpha, Portable ISA, ARM, and x86). Amongst its available simulators, the *sim-outorder* is a cycle-accurate simulator for a single out-of-order executing processor. SimpleScalar simulates only user-mode execution and uses emulation for I/O operations and system calls. The importance of such system effects is investigated in PHARMsim [25], which is a *full-system simulator* for the PowerPC ISA. Full-system simulators simulate the entire software stack covering both application and OS-level effects. In PHARMsim this is accomplished, by simulating OS operations and utilising a *DMA-engine* to determine the effect of cache state errors on disk I/O activity.

TFSim [90] is a full-system simulator with decoupled functional and timing simulation concerns, which introduces the *timing-first* paradigm: each instruction is first simulated with an integrated "timing" simulator, which models out-of-order execution and predicts the interleaving of threads. To improve its performance, this timing simulator does not simulate aspects of the system with low performance cost. As this might reduce the simulation precision, committed instructions of the timing simulator are forwarded to a full-system functional simulator (like Simics [84]), which detects deviations in the simulation process and, if required, triggers recovery actions. The timing-first approach is used in the General Execution-driven Multiprocessor Simulator (GEMS) [89] framework, where the timing aspect is offered by a flexible multiprocessor memory system simulator.

PTLSim [149] is a coupled full-system simulator, that supports cycle-accurate functional modelling for the x86-64 architecture. Similar to the SimpleScalar paradigm [6], the PTLSim offers a superscalar out-of-order core, a simultaneously multi-threaded version of that core and an in-order sequential core, along with cache memory models and TLBs. TLB miss delay is assumed to be proportional to a chain of four dependent loads (one per level of the 4-level x86-64 page table tree). Incoming interrupts and DMA requests are handled with trace recording and injection: a timestamp is assigned to each such operation, which will only be issued if the *simulation time* reaches that timestamp. PTLSim can simulate multiprocessors in parallel, by assigning a core model to each core, advancing each core by one cycle in round robin order and handling shared memory operations.

Instruction-set simulation

Instruction-set simulation (ISS) reduces the overhead of simulation, by focusing on the instruction-set architecture (ISA) of the target host. Such approaches sacrifice the flexibility of *CAS* for performance, by abstracting the inner functionality of the CPU. The contents of registers are still correctly accounted for, thus rendering instruction-set simulators (like Simics [84]) useful for debugging purposes.

Simics [84] is a full-system simulator which simulates both workloads and operating systems on a variety of target architectures. It can model embedded systems, desktop or set-top boxes, telecom switches, multiprocessor systems, clusters and networks of all of these items. Simics views each target machine as a node, representing a resource such as a Web server, a database engine, a router or a client. A single Simics instance can simulate one or more nodes of the same basic architecture, while heterogeneous nodes can be connected into a network controlled by a tool called Simics Central. Simics offers accurate instruction counts, but only estimates the interleaving of memory operations of next generation systems by including approximate cache and I/O timing models.

ISS is used for system-level design of embedded applications, where the CPU cycles of each operation on the target host are instrumented to the application code, before it is simulated. Cache- and memory-related operations are typically simulated. Bammi et al. [12] propose compiling the source to a virtual instruction-set and generating a processor basis file that contains an estimated cycle count for each virtual instruction. The cycle counts for each instruction are either defined from processor manuals or through cycle-accurate simulators. Alternatively, statistical methods are used to determine the cost of each virtual instruction, based on execution times of benchmarks with known instruction mix. However, this method does not take into account compiler optimisations. To overcome this, Lazarescu et al. [77] use an assembly-to-C translation approach to add timing information on the target binary, after the target compiler applies its optimisations.

Wang and Herkersdorf [139] propose an intermediate instrumented source code, that adds

timing information at the basic-block level, after the front-end compilation stage. Timing information is defined statically for local effects of basic blocks and dynamically for global effects like memory accesses, whose cost depends on the context. A similar hybrid approach is followed by Meyerowitz et al. [93] for heterogeneous multiprocessors, where communication delays between processors have a predefined cost.

Another level of abstraction is offered via *instruction-set emulation*, which emulates the execution of each instruction on a target host with a set of instructions on the simulation host. This typically involves emulators like QEMU [13] or the binary translation module of SimOS [117]. Dynamic binary translation is used in Embra [145] for fast full-system simulation of MIPS R3000/R4000 processors, Mambo [137] for PowerPC and COREMU [140] for x86_64 and ARM architectures.

The Structural Simulation Toolkit (SST) [115] is a configurable and extensible discrete-event simulator for MPI-based HPC applications. The SST uses multiple processor, memory, I/O and network models to investigate the performance, power consumption and reliability of future large-scale systems. Specifically, SST has been introduced with a CPU model based on SimpleScalar [6], DRAMSim2 [118], DiskSim [2], QEMU [13] or even a stochastic CPU model. The decision on the selections between these models, determines the granularity of simulator and its position in the time-accuracy tradeoff.

Using a different approach the Virtual Timing Device (VTD) [63] decouples functional and timing simulation. The code is simulated on the QEMU functional virtual platform and the timing aspect is provided by a variety of mathematical models, which differ in terms of precision and accuracy. These timing models combine linear regression on the event counters of the functional simulator with system components simulators. The simulation alters the timestamps returned to the application, but does not affect the sequencing of thread events in the functional virtual platform. VTD returns less accurate predictions than other approaches (errors of 18%) at a very low overhead of less than 10x. The simulation takes place on a single host.

Direct execution

Direct execution is a technique used to speed up simulation, by executing the target program instructions on the simulation host, assuming that the ISAs of the target and simulation hosts are the same. This type of simulation is used to investigate parallel architectures in terms of shared memory systems, node interconnect etc.

The Rice Parallel Processing Testbed (RPPT) [32] is a typical example of direct execution execution-driven simulation. In the RPPT, the code of a concurrent application is instrumented to dynamically generate the workload that drives a simulation model of a parallel architecture. The application is divided into basic blocks of execution, i.e. blocks of code without interaction points between processes. The instruments also include a fixed timing of each block, acquired by its execution on a single-processor host. Interaction points can be accesses to shared memory or messages between processes. Each basic block is directly executed on the host of the simulation and its fixed timing is added to the simulation time. The way that the simulation time progresses is defined by a user-provided model that maps processes and data on the architecture model. The architecture model is also used to simulate the cost of interprocess communication. As parallel processes are serialised and executed on the same host, time and memory issues affect the scalability of the approach.

Tango [35] follows a similar methodology to the RPPT to offer multiprocessor simulation on a uniprocessor. Each processing element in Tango is a different process, which is instrumented at the basic block level to update the local simulation clock of the process. Basic blocks are defined in a flexible manner, as users can decide which operations are considered interaction points: only synchronisation points between processors, only shared data accesses or both. By allowing these three different configurations Tango offers a "pay only for what you need" approach that reduces simulation times if only synchronisation points are needed. However, Tango suffers from higher overheads due to the use of processes and semaphores.

PROTEUS [21] was proposed as a solution to the shortcomings of Tango, by using lightweight threads instead of processes. PROTEUS offers a modular approach, allowing for usage of various

OS kernel, shared-memory, cache and network modules: each related instruction accesses one such module, which determines the functional and timing behaviour to be simulated. These traps are also used for synchronisation reasons between the different simulation threads. The synchronisation mechanism is augmented with a quantum-based technique that guarantees that no thread is much further ahead in simulation time than the others. For the parallel sorting benchmarks presented, PROTEUS achieves very good accuracy at a typical overhead of 35x to 100x, compared to the 500x to 2,000x reported in their paper for Tango.

The RPPT, Tango and PROTEUS approaches simulate multiprocessors on a uniprocessor. The Wisconsin Wind Tunnel (WWT) [113] uses a parallel computer to simulate shared memory multiprocessors, in an attempt to improve simulation time, exploit the inherent parallelism of parallel applications and reduce the stress on the memory requirements of the single host. The WWT exploits features of the hardware and modifies the OS to simulate behaviour only in the case of a cache miss; any other code is directly executed on a processor. Parallel processes advance in time-quanta, synchronising either at time points when events that require simulation (e.g. cache misses in shared memory architecture) are triggered. If no such events are triggered for Q cycles (that correspond to a quantum), then the processor blocks at a barrier, waiting to synchronise with the other ones. Instrumentation is used to locally account for these Q cycles in each process. In an attempt to improve the portability of the WWT approach, [98] presents the Wisconsin Wind Tunnel II, which employs binary instrumentation and an architecture-neutral programming model to replace platform specific requirements. However, this comes at the cost of higher overhead compared to the WWT.

The Warwick Performance Prediction (WARPP) toolkit [58] introduces a semi-automatic discrete-event simulation technique to provide performance predictions for large scale scientific parallel applications. The idea is to combine instrumentation at the basic-block level with a scripting language that describes the control flow of the application to generate a simulation model that is then used to estimate the performance of a future architecture. I/O timings modeled as "times per byte" and network profiles of different regions of the system under simulation described by bandwidth and latency are also input to the simulator. The simulator then starts a number of virtual processors which progress the various control flows of the program, processing

the different events that include CPU time, idle time and I/O or network activity. WARPP scales well offering good predictions to simulations of thousands of nodes. However, in order to achieve this scalability, it does not model the synchronisation interactions between threads or the changes in the profiled latencies as a result of the contention on the system's resources.

Shared memory multiprocessors are also the research subject of the Graphite [95] approach, which simulates the execution of multithreaded applications for target multicore architectures on a distributed system. Each thread of the application is mapped on a host of the distributed simulator and its functional behaviour is "simulated" by direct execution. Graphite uses dynamic binary instrumentation with Pin [111] to offer a single memory address space and consistent OS interface amongst all nodes. Instructions and events from the core, network and memory subsystem functional models are passed to analytical timing models that update individual local clocks in each core. Although Graphite provides a modular approach to enable the change of the tradeoff between performance and accuracy, typical simulation overheads have a median of 1200x for the simulation, which compares well to other full-system simulation approaches.

BigSim [152] is a parallel simulator built on top of the CHARM++ runtime system that targets on predicting the performance of machines with a very large number of processors. The proposed approach involves the real execution (functional emulation) of a parallel message-based application on a number of nodes, whilst running a parallel algorithm that corrects the timestamps of the messages sent between the simulated processes. Three ways are supported for modelling the timing behaviour of the sequential code executed on each node of the simulator: user supplied expressions for every block of code, scaled wallclock measurements to match the running time of the target machine or hardware performance counters to be used as inputs to heuristics that determine execution times. Network performance for the communication of the nodes is either predicted via mathematical models or with a network simulator.

COMPASS [7] is another direct execution simulation method that focuses on the simulation of MPI-parallel applications. The framework natively executes instructions of different processes and traps MPI I/O and network communication. The timing of I/O operations includes simple

models based on disk parameters and a detailed disk model, while the network communication uses a contention-free latency-based timing model. The physical times of such operations, which are executed for correctness, are disregarded, thus decoupling functional and timing concerns of the framework. The basic component of COMPASS is the kernel, which schedules threads and conservatively synchronises cores that are simulating in parallel. Presented results for COMPASS indicate prediction accuracy of within 5% and highly variable overhead, ranging from 2.5x to 40x, depending on the application and host processors used.

Parallel simulation

Simulation exhibits a clear tradeoff between speed and accuracy. The overhead of a full-system cycle-accurate simulator can be several orders of magnitude higher than direct execution, reducing the applicability of simulation for heavier workloads. Parallelizing the process has been an essential way to speedup the process, whilst maintaining the required accuracy [113, 152, 29, 106]. Various synchronisation techniques have been proposed to coordinate the nodes of the parallel simulator.

SlackSim [29] offers a platform for fast parallel simulations. The POSIX threads of a multi-threaded application are distributed amongst the cores of the simulation host, where they each execute an integrated CPU simulator of the target architecture (pipeline and L1 cache). The L2 cache banks and their interconnection to cores are simulated by a central manager thread. Memory management, file system handling and other system functions are emulated outside the simulator. Each core has its local clock, incremented according to the simulator, and a local maximum timestamp, which determines how far the thread can progress the simulation without suspending. SlackSim offers thus a coupled low-level instruction simulation for each core. A global simulation thread is responsible for synchronising the threads, by setting the maximum local timestamp of each core, according to the synchronisation scheme followed. Apart from the trivial cycle-, N-cycle- (quantum) or no- (lax) synchronisation schemes, SlackSim proposes a *bounded slack simulation* scheme. This is a moving window approach that forces threads to synchronise only if they exceed a difference from the constantly updated global simulation

time. SlackSim offers very accurate results (less than 3% error), but at a significant overhead.

Graphite [95] simulates a multithreaded application on a distributed system. In order to keep the local clock of each node synchronised with the rest it supports three types of synchronisation: *lax*, where all threads progress freely, synchronising only when they interact to each other, *lax with barrier*, where they all synchronise after a configurable number of cycles (quantum), and *lax with point-to-point*, where each node synchronises with another random node.

BigSim [152] adapts the traditional optimistic Parallel Discrete Event Simulation (PDES) paradigm, by modifying the language and runtime support, to identify and exploit the determinacy of parallel programs and reduce the cost of synchronisation. This means that event dependencies are recognised on the source code level and an event can only be allowed to execute after all events that it depends on have been executed. This accelerates the synchronisation scheme as the system avoids checkpointing and rollback costs.

2.2.2 Network emulation

Network emulation deals with the integration of real-time application prototypes and network simulators for network protocol evaluation. As the prototypes need to execute in real time and the network simulators in simulation time, the problem is to integrate the two and enforce a correct ordering between the real events from the applications and the simulated events from the simulators. Apart from the functional correctness of the emulation, each proposed approach for network emulation is concerned with prediction accuracy, overhead and scalability issues.

A first attempt to network emulation is provided by Dummynet [114], which hacks the TCP/IP protocol to intercept network communication of unmodified applications within the OS kernel. The intercepted packets from node A to node B are queued and routed, according to the network from A to B that we want to simulate. However, such delays are enforced in real time, thus limiting the applicability of the method based on the timing granularity of the simulation host.

Using virtual machines on top of a single physical machine constitutes a common way to scale-

up the emulation capabilities. Within this framework, a number of solutions have used virtual time as a means to alter the perceived time duration by the virtual machines participating in the emulation and thus further increasing its scalability capabilities. Gupta et al. [57] propose the *time dilation factor (TDF)*, which leads virtual machines to perceive external events as if they occurred much faster than in reality, thus replacing real time with virtual time. Changing time perception on the Virtual Machine Monitor (VMM) level, namely on the software layer responsible for multiplexing many virtual machines on the same physical resources, the authors achieve pervasiveness and transparency: all virtual machines will be unaware of physical time while no changes are required on the application level. By using real time delays on the VMM-level, the emulation running time will be increased as many times as the highest TDF value of the monitor. This increases the overhead of the method. Another problem is the low hardware utilisation resulting from the stability of the TDF.

In order to alleviate this issue, Grau et al. introduced the time-virtualised emulation environment [54], which dynamically changes the TDF according to some load metric to better balance the emulation's resource utilisation. Nevertheless, in both cases the passing of time is altered at the level of a virtual machine and for all its components. This results into poor granularity in the level of alteration and restricts these techniques from uses beyond network emulation like extending or accelerating the duration of an application method to identify its performance effect.

Time-dilation is also used in the Scalable Virtualised Evaluation Environment for TCP (SVEET) [43], a framework for large-scale TCP experimentation. Distributed applications are executed by virtual machines, that have any TCP communication trapped towards a global network real-time simulator. By selecting a single TDF factor for all virtual machines, SVEET virtualises the load rate to the network simulator, allowing large-scale, flexible experimentation.

Bergstrom et al. [14] propose the distributed Open Network Emulator (dONE), which integrates direct code execution of unmodified network applications with network emulators/simulators. They introduce the idea of scaling real time to allow enough time to the physical network to emulate packet transmissions. The event timings of the transmissions are determined by

a network simulator. This time-scaling approach is defined as *relativistic* time. Relativistic time also enables time warps over quiescent intervals amongst all virtual hosts. A central coordinator is used to determine *epochs*, during which real time progresses under the same time-scaling factor. Therefore, time keeps flowing normally (as in physical time), but remains controllable in order to be able to include times from the simulator's discrete events. The composition of unmodified applications and protocol stacks is offered by the Weaves framework [136], while the Linux kernel is customised to enable logging of virtual times. The approach assumes that only the API to the network protocol stack needs to be considered as an external event; any other time progress is local and should be entirely taken into account, since no instrumentation overhead is added. This makes relativistic time (as presented in [14]) less applicable for profiling or time-scaling at the method level.

The SliceTime [143] platform decouples wall-clock time from the perceived progression of software prototypes' time, allowing network simulators and prototypes to be synchronised in virtual time. A timeslice approach is enforced by a central scheduler to enable parallel hosts to progress in a synchronised way and let the network simulator process only the events within that timeslice. The objectives and synchronisation patterns of SliceTime are quite similar to our approach. However, the implementation is based on executing software prototypes within virtual machines, while modifying the hypervisor layer to virtualise the passing of time. Note that individual method executions within these hosts cannot be resolved and thus it is not possible to create a per method time mapping on the virtual time dimension.

2.2.3 Real time simulation

In this section we review existing research that focuses on simulating hypothetical method durations or performance models on the method level in real time (and not in virtual time like this thesis). Fine-grained performance time-scaling is offered by the dynamic performance stubs approach [132], which replaces existing code with a functionally similar stub, that simulates the performance behaviour of the original code. Based on a calibration run to profile the host and an initial profiling run to identify performance characteristics of methods involved,

the methodology replaces functions with stubs. Modifying the performance behaviour of stubs and simulating this effect on the system (e.g. with busy waiting or sleeping), this approach addresses specific performance questions to support software optimisation. However, thread-level interactions (e.g. synchronisation points) of the original code with the remaining application are not reproduced in the real time simulation.

Dunn [39] simulates performance models in real time, as a means of replacing missing methods, during the early software development stages, to enable performance testing through the lifecycle. Assuming the existence of performance models for unimplemented methods, then whenever such a method is invoked, the running thread is set to sleep or busy wait for an amount of time determined by sampling the corresponding performance model. As implemented methods consume CPU cycles as well, this approach integrates models and code in real time, leading them to contend for the CPU. Similarly to the approach found in [132], locks or I/O contention between different threads "executing" the same model in parallel is not addressed. Moreover, the simulation has to occur in real time, increasing performance testing time.

2.2.4 Controlling program execution in Virtual Time

Virtual Time (VT) is any perception of real time that is different from the physical duration of that amount of time. This includes stretching or shortening time periods, accounting for time in any arbitrary way, synchronising with times that do not necessarily correspond to the real ones or even shifting the occurrence of events backwards in time.

The idea of Virtual Time was initially introduced in [70] as an optimistic synchronisation scheme for distributed systems. Processes are allowed to speculatively continue execution, disregarding possibilities of synchronisation faults. If such a case is identified, the process has to *roll-back*, thus "returning" the Virtual Time acquired for the speculative execution and moving back in time. This main idea was later evolved in [69] and [112] to the Time Warp operating system that supported rollback mechanisms at the system level to facilitate optimistic event scheduling for parallel discrete-event simulations on multiprocessors. Chandy and Lamport use the notion of Virtual Time (denoted though as process local time) to acquire consistent global snapshots

in distributed systems [27]. In this work, local clock values that are synchronised only on inter-process communication constitute virtual times, since the timings of events that do not affect causality in the distributed network might acquire random (though still sequential within that process) values.

In [107], the Pin [111] binary instrumentation framework is extended to define a user-level thread library to control scheduling through instrumentation. A scheduler controls the execution of the application's threads, by adding checkpoints and mapping at each of them the previously stored thread contexts to a single user thread. This simulates context switching, allowing the thread scheduler to create custom thread-interleavings as our scheduling profiler does. Any timing information (including time dilations) is provided by a system simulator. Without an external simulator the thread library cannot simulate the contention on the I/O subsystem from parallel threads, since all threads are mapped to a single-thread. Thread control is offered at checkpoints potentially creating a discrepancy between real system scheduler behaviour and simulation one.

A prototype of the virtual time execution for uniprocessors was first introduced in [18]. Dividing the execution of a Java application in basic blocks, this approach followed closely the paradigm of direct execution execution-driven simulation [32, 21, 35]. In this approach the scheduler of the simulation is tightly coupled with Java and uses JVMTI calls to synchronise the application's threads. POSIX signals are used to poll the CPU times of threads. This combination of JVMTI and POSIX at the native level leads this prototype implementation to suffer from "spurious deadlocks", due to synchronisation effects within the JVM. No I/O contention, multicore targets or integration of performance modelling are supported.

2.3 Java Virtual Machine overview

2.3.1 Java

Java is an object-oriented programming language specified to achieve platform-independence by executing programs on a virtual machine (the Java Virtual Machine - JVM), which is built on top of any underlying platform and operating system. The JVM only recognises and executes bytecode, namely a set of instructions each codified in an opcode of 1 byte length. User source is compiled to Java bytecode, which should then be executed on the JVM on any platform. Java's ubiquity, instrumentation capabilities and usage in a variety of widely industry-deployed applications like Web and Application Servers, have comprised the contributing factors towards the selection of this language as the target of this thesis.

2.3.2 JNI

The Java Native Interface (JNI) [130] concerns the capability of Java programs to include libraries written in native code, i.e code executable on a particular platform. Although this approach limits the scope of Java code platform-independence, it allows usage of existing code (written in C or assembly) or system libraries within a Java application, thus limiting the extensive need for code re-writing from the one language to the other.

2.3.3 JVMTI

The Java Virtual Machine Tool Interface (JVMTI) [129] regards the features initially introduced in Java 5 used to facilitate the generation of Java debugging and profiling tools. This is accomplished by a JNI-written agent library, which is defined as an argument (-agentpath) on the JVM execution. The agent library interacts with a Java application on a number of predefined events (*callbacks*), like class loading, method calling, monitor entering, garbage collecting etc, offering instrumentation capabilities on each of these events. JVMTI is typically

used for debugging and profiling purposes.

2.3.4 Java instrumentation

Java instrumentation regards the series of techniques that automatically modify existing classes, like adding time accounting code in the beginning and end of each method. Such methods are provided since Java 5 by the classes of the java.lang.instrument package. Defined as a Java-agent (-javaagent JVM argument), a class derived from this package may be used to transform other class files as they are loaded. Tools like ASM [22] or BCEL [33] are typically used to offer such transformation capabilities.

2.4 Pthreads

Threads are streams of program instructions that are handled independently by the operating system. A program comprises one or more threads, which have their own stack frames, but share the heap. This allows for multiple threads to process different parts of memory in parallel, thus utilising the available processing resources and improving the performance of a program. In this section we introduce the POSIX threads (Pthreads) implementation of the threading paradigm, which is a well-established API designed for portability.

The lifecycle of a thread starts, when it's parent process spawns it. This process initialises the context of the thread with the routine it is supposed to execute and sets it into the queue of runnable processes. The thread is then resumed when the operating system scheduler decides to do so. POSIX uses the *pthread_create* function to spawn a new thread. The parent of the thread may invoke the *pthread_join* function to block until its child thread terminates execution.

The sharing of the heap between different threads introduces the need for synchronisation. Generally, this is accomplished via mutual exclusion mechanisms, according to which one thread acquires exclusive access to a part of memory. Any other threads that attempt to access in the

meantime are blocked until the exclusive access requirement is revoked. Pthreads introduce the following functions to achieve this type of synchronisation:

- *pthread_mutex_lock(pthread_mutex_t*: acquire a mutual exclusion variable. If another thread attempts to acquire it while it is being held, then that thread will be blocked.
- *pthread_mutex_unlock(pthread_mutex_t*: release a mutual exclusion variable. If another thread is waiting at a *pthread_mutex_lock* invocation then it is allowed to resume and acquire the variable itself.
- *pthread_cond_wait*: a thread that has already acquired exclusive control of a mutual exclusion variable, relinguishes the control and waits on a *conditional variable* to be notified by another thread on when it should continue. Once this happens, the currently waiting thread will attempt to acquire the mutual exclusion variable and resume execution.
- *pthread_cond_signal*: a thread signals another thread that is waiting on a conditional variable to resume execution when the associated mutual exclusion variable becomes available.

These operations guarantee that the state of the mutual exclusion variable can be read and updated without the need of further synchronisation. This means that if two or more threads try to acquire the mutual exclusion variable "at the same time", only one of them will achieve to do so.

2.5 I/O

2.5.1 I/O in Linux

I/O devices concern a variety of media: hard disks, disk arrays, DVDs, network cables, printers, monitors, terminals, mice etc. They are generally distinguished in block devices, i.e disks of any kind, and character devices, like terminals and mice, where I/O operations concern sending and

receiving streams of bytes. Linux has a unique interface for all types of I/O devices, accessing them via file system calls. Nevertheless, the performance behaviour of I/O operations differs significantly between block and character devices or amongst character devices themselves.

Accesses to block devices typically bear the highest overheads, due to the high volume of data that get transferred. The high-level handling of the Linux kernel for block I/O operations is shown in Figure 2.2. It employs several optimisation techniques, like caching, I/O scheduling and Direct Memory Access (DMA) support. Directory entries, file information and disk pages are stored in the *dentry*, *inode* and page caches respectively, improving the performance of lookup and retrieval of file blocks. In case that the requested data are not in the page cache, a new I/O request is generated and enqueued as a pending request. To exploit the fact that sequential disk blocks are accessed faster, an I/O scheduler determines the order that pending requests are issued to the device driver. By merging I/O requests to consecutive blocks together, Linux limits the number of issued requests. Keeping the CPU idle (or in a busy-loop), while waiting for I/O is not suitable for disks, whose transfer latency is typically in the order of milliseconds. Instead, DMA allows the CPU to program a special-purpose external controller to perform the I/O operation, while the CPU is assigned to another runnable process. When the I/O operation is finished and the block transferred to memory, the DMA controller sends an interrupt to the CPU, to set the originally issuing process as runnable. Modern disks use on-board buffering as well, to avoid rotational delays.

Character devices like mice and keyboards only transfer a few bytes per operation and thus bear low run time overheads. In contrast, sound cards and network controllers use buffering, filling a cyclic buffer with data and using DMA when more data are requested or data are available for reading. They also differ in that sound cards use standard rate in their streaming of data, while network arrivals are asynchronous.

2.5.2 I/O in execution-driven simulation

Execution-driven simulators handle the issuing of I/O operations via emulation, modelling or "virtualisation". We note here that we are not interested in I/O simulators (like DiskSim

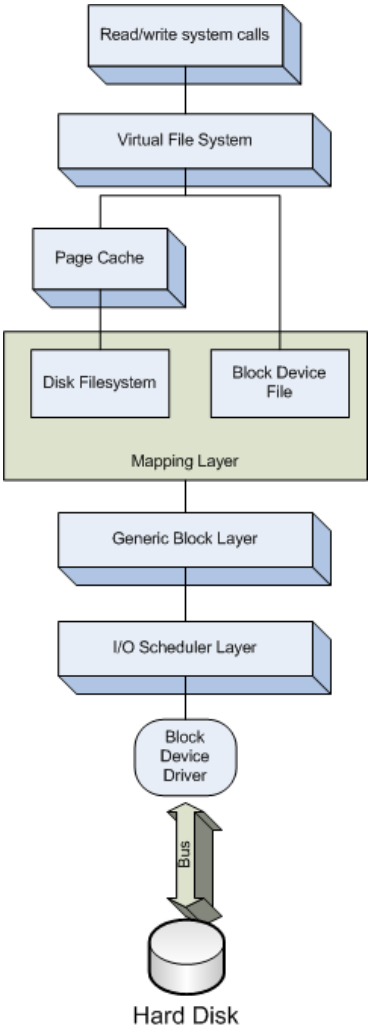


Figure 2.2: Linux kernel components involved in block I/O (Based on [20])

[2]), but rather higher-level system simulators that have an I/O component. SimpleScalar [6] emulates I/O operations and system calls, executing them directly on the simulation host. PHARMsim [25] introduces a *DMA engine* that models disk I/O latency, to include the effect of disk activity to cache coherency in the simulation. Network traffic is simulated via mathematical models [152] or via network simulation [152, 5]. Simple or detailed disk- and a contention-free latency-based network-timing model are used in the COMPASS [7] simulator for parallel MPI applications. PTLSim [149] virtualises the occurrence of I/O operations on the VM hypervisor level, by assigning a timestamp to each incoming interrupt or DMA request. Only when the simulation time reaches this timestamp, will the event take place.

2.5.3 Prediction methods

I/O operations are random by nature, depending on the context of execution, optimisations used [62], mechanical considerations like the current position of a disk head with respect to the block to be transferred, events like new request on the network card, or even human factors. The main focus of our approach is on modelling and predicting at runtime the observed time of each I/O system call, including kernel file-system management, I/O scheduling, page caches, DMA-requests, bus and disk or network latencies.

An attempt to model the entire I/O subsystem is introduced by Nguyen et al. in [101]. The authors model I/O caching effects, using a stochastic Petri Net for the read and write operations in the ext3 Linux file system and implement the I/O scheduler and disk components of the model using a simple queueing node. The model's parameters are populated using the *iozone* benchmark. This approach requires a model calibration and is file system specific, contrary to our method that treats the I/O subsystem as a black-box.

Black-box models provide a flexible and easily applicable way to predict I/O performance, as they do not assume any knowledge for the underlying system. Anderson [3] populates a table with I/O measurements related to parameters like storage array information and I/O request types and then returns a prediction of the maximum throughput for an arbitrary set of input parameters. Similar parameters are used to induce probabilistic models of storage arrays in [74]. Wang et al. [138] employ machine-learning techniques using as parameters the arrival rate, the logical disk block number, and the I/O request type and size. Black-box models based on regression trees have been introduced for solid-state drives in [79].

Garcia et al. also simulate disk drive times in [50]. Using both real and synthetic workloads, they gather measurements of the disk service times to produce a variate generator that returns histogram samples. Three different types of histograms are supported: a global one, one for each I/O operation (i.e. `read` and `write`) or one per combination of I/O-operation and size of bytes transferred. The fact that the profiling run precedes the analysis and generation of simulated samples renders this approach unsuitable for I/O prediction at runtime.

The same applies for the work presented by Shan et al. [120] populate parameters like API, file size and parallel tasks of a configurable I/O stress benchmark, according to the characteristics of the application under study. This allows them to execute the benchmark and acquire I/O time predictions. Using the same benchmark Meswani et al. [92], collect statistics regarding I/O operations on various hosts. By statically instrumenting the I/O calls of HPC applications, they gather and compare the profiled I/O durations to the ones of the benchmark for the base system. The ratio between the two is then used as a weighting factor on the benchmark's results of the target system, yielding performance predictions for the total I/O time on that host.

Zhang et al. [151] use time-series based predictors, to estimate the system resource consumption (including I/O times). As the precision of the prediction model is workload-dependent, this paper investigates the usage of classification algorithms like PCA and k-Nearest Neighbors to forecast the best system resource performance predictors. This approach allows the forecasting of the best predictor (amongst the available one) for the workload under study. However, it also requires a training execution to generate the best predictor database.

I/O analytical models that cover effects like prefetching, caching and scheduling on top of disk latency are introduced in [122]. The formulas of the model are populated by workload characteristics and disk parameters, and their predictions are compared to real time durations, to effectively identify performance anomalies in the Linux kernel. The approach is extended in [121], where Shen et al. define a reference performance behaviour for a system, by utilising *change profiles* that probabilistically characterise its expected performance deviations. Empirical results are then compared to this reference, in terms of how likely it is for the system to deviate enough from the reference value to provide the measured one. Extreme deviations manifested as low p-values are identified as performance anomalies. Although these models can uncover performance anomalies on the Linux I/O subsystem via statistical analysis, it is not clear how accurately they can predict the duration of the next I/O request at runtime.

Chapter 3

Virtual Time Execution

The key idea that underpins this thesis is to execute an application in the form of an execution-driven simulation (covered in Section 2.2.1) to provide empirical predictions to hypothetical performance questions without resorting to lower-level system simulation [6, 26, 2]. In contrast to other execution-driven simulation approaches [32, 35, 21, 113] that add instruments to measure clock cycles between process interaction points statically, our aim is to directly execute the threads of the software under test and profile them on the simulation host hardware at runtime. We define a *virtual time execution* as the approach that maps execution times to virtual times via time-scaling or replacement with a time prediction from a performance model and then uses these virtual timestamps to determine the sequence in which threads are scheduled. This may affect both the functional and timing behaviour of the program. From a performance testing perspective, a virtual time execution tool can be perceived as a *scheduling profiler*: a profiler that also controls the scheduling of the program it is monitoring.

The difference between a real execution and a hypothetical virtual one is shown in Fig. 3.1. An application with two threads T_1 and T_2 is executed on a uniprocessor machine and the two threads compete for the acquisition of the same lock to protect a critical section (bold part of the line). In the real time execution and assuming a fair scheduling policy, T_1 acquires the lock first and is then context switched to allow T_2 to make progress. T_2 is blocked waiting for the lock to be released. After this happens, the real time scheduling results into both

threads issuing I/O requests in parallel. Now consider that we accelerate by a factor of two the execution of the method that T_2 runs before requesting the lock. This produces a new thread interleaving, where T_2 acquires the lock first, releases it within the same timeslice, and allows T_1 to make progress immediately. Moreover, T_2 is now the first thread to issue an I/O request, that finishes before T_1 issues its own. The virtual time program execution simulates correctly this new behaviour.

The example of Fig. 3.1 highlights some important aspects of virtual time execution. The total execution time of T_2 has been altered by a factor that is higher than the proportion of the accelerated part, due to the elimination of the waiting time in the "post-optimisation" virtual schedule. The load and the sequence of requests to the I/O subsystem has also changed, affecting the resource access patterns and its response time. The overall effect is a change in the execution characteristics of the code which, in cases where there is queueing for resources, may lead to non-linear changes in queueing times and thus in the overall application performance. Such effects may be hard to predict with simple 'back-of-the-envelope' calculations.

Time-scaling effects on unmodified applications have been studied as a means to scale network emulation tests beyond the limits of available hardware [57, 56, 54, 14]. Typically scaling occurs at the virtual machine level [57, 56, 148] and in an epoch-based manner [54, 14]. In contrast to these methods, virtual time execution offers finer-grained analysis capabilities, by allowing users to scale application threads at the method level and simulating the effects of scaling on the thread schedule. The virtual time execution methodology also poses entirely different challenges, as enforcing the thread schedule and adding profiling instruments at the method level directly affect the performance behaviour of the profiled execution.

We clarify here that figures throughout this thesis that present virtual (or real) time executions always assume a fair scheduling policy: processes/threads and cores are assumed to progress at a timeslice of exactly s and the background load of the system is disregarded. Though unrealistic as it might be in light of preemptive context switches, exception handling, scheduling of threads with higher priorities etc, this "sterile" environment allows us to determine a correct order of execution and expect a particular sequence of thread events (like the lock acquisition

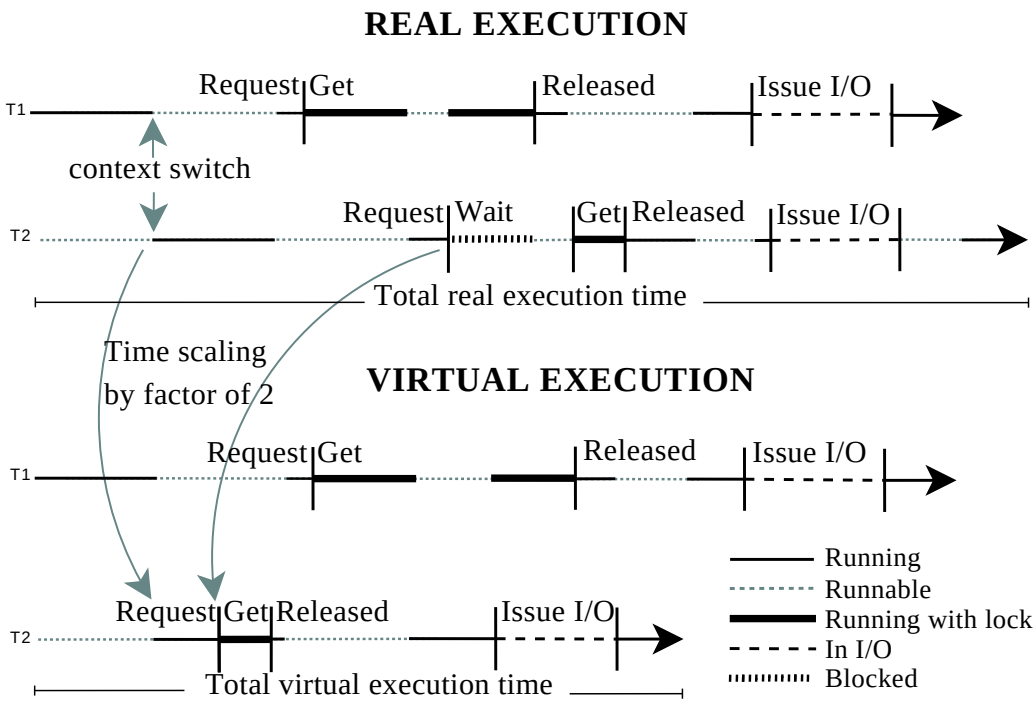


Figure 3.1: Real- vs Virtual Time Execution

of Fig. 3.1). In turn, this enables us to show how the virtual time execution affects or simulates such behaviours. A schedule that generates the expected order of our isolated environment, when executing a program in virtual time is defined as a *correct virtual schedule*.

Applying a correct virtual schedule, instead of the real one, may alter the sequence of events and thus lead to a different functional behaviour, for example, by affecting the sequence of file I/Os. Clearly, dependence on the order of system events contradicts sound software design principles, that require no assumptions on the order in which threads make progress in non-critical regions. It would be possible for VEX to be modified in order to identify such "buggy" behaviour, by enforcing various virtual time schedules and detecting differences between them. However, this is beyond the scope of our research.

Broadly, performance engineers can use virtual time simulation in two scenarios:

1. During the development phase, performance models describing incomplete components are simulated in virtual time together with the implemented parts. The output of the virtual time execution is an estimate of the performance of the end product, according to the provided models and code implementation. In this case, the accuracy of the estimation

relies on the quality of the models provided for the components.

2. At the completion of the implementation stage, hypothetical scenarios for the execution times of components can be performance-tested in virtual time. For example, this approach could lead to faster bottleneck identification and better optimisation choices.

The hypothetical scenarios are acceleration or deceleration factors for execution times, called *time-scaling factors* (*TSF*s), or estimated performance results acquired by a performance model. The former specifications concern already-implemented code and provide answers to "what-if" questions. The latter facilitate the simulated execution and profiling of partially complete code. We refer to such hypothetical scenarios as *virtual specifications*.

In this chapter, we present the design and implementation details of our prototype virtual time simulator, called the Virtual Execution Framework (VEX). VEX is a C++ shared library for Linux. It is language independent in that its functionality is decoupled from any particular programming language and it is self-contained, as it assumes no specific programming system like other execution-driven simulators that rely on MPI [7, 14] or Charm++ [152]. Its main objectives are thread scheduling and virtual time profiling. They function on the user-level and can be utilised by a program under test, through an API that is typically invoked via code instrumentation. In this presentation of VEX we assume that the language-specific instrumentation upper layer exists and provides the information required by the framework. In Chapter 5 we will present the corresponding layer for Java.

The design of VEX is shown in Fig. 3.2. The framework has three main components: the scheduler, the profiler and the I/O module. The former two are presented in this chapter and the third one is discussed in Chapter 4, as it deals with the research issue of empirical I/O prediction, which is somehow distinct from the core VEX methodology. We note here that Fig. 3.2 omits the performance model simulating component, which is considered an extension of the base VEX. This is presented in Chapter 8.

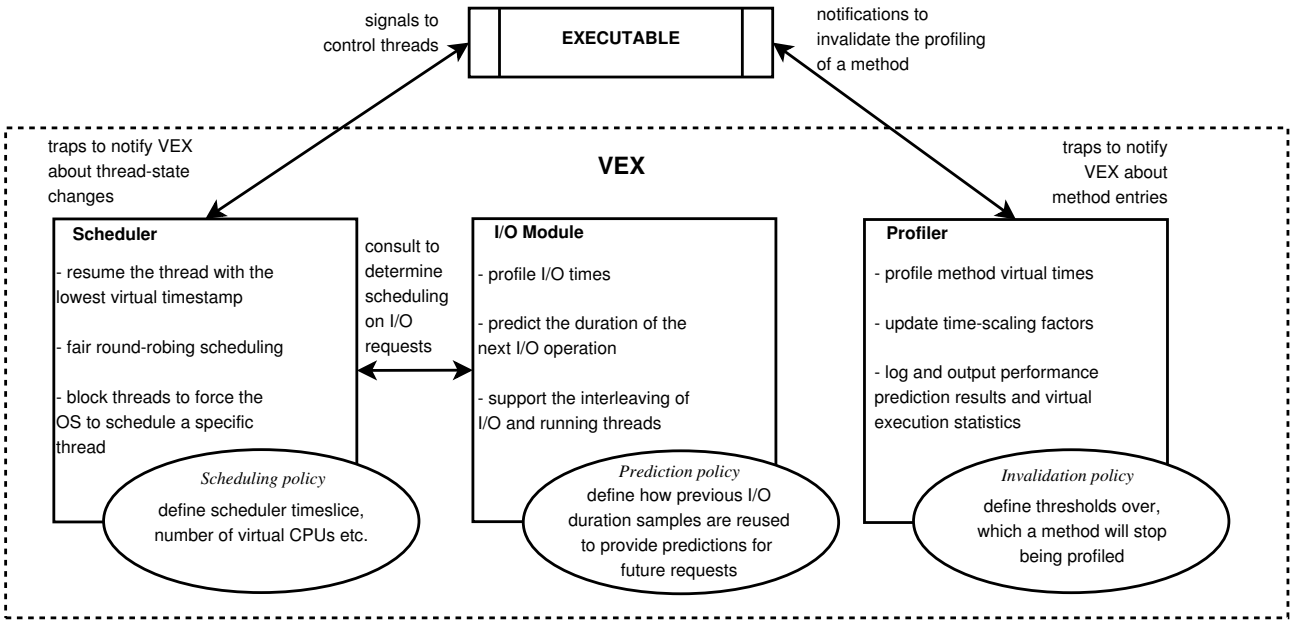


Figure 3.2: VEX design

3.1 Scheduler

The virtual time scheduler enforces a priority-free round-robin scheduling policy, with a default virtual timeslice of 100ms (which is the base time quantum for the default static priority in Linux [20]). The next thread to be resumed is the one that was suspended earliest on the virtual timeline; this is essentially the *Borrowed Virtual Time* (BVT) scheduling policy, as described in [38], with our method time scaling factors being equivalent to "weights" of BVT processes. Our objective is to provide a simple "fair" priority-less scheduler that broadly approximates the typically much more complex scheduler of the underlying operating system. However, there is no inherent restriction on the scheduling policy that the framework enforces.

The virtual time scheduler works on top of the operating system scheduler and is executed in user-space. Using low-level condition variables it blocks all threads except the one that it wishes to run. It then relies on the underlying operating system scheduler to schedule that thread using its own policy. Crucially, the underlying operating system scheduler has no other choice but to run the application thread determined by the VEX scheduler.

3.1.1 VEX states

In order to identify the next runnable thread to be resumed and handle thread communication in virtual time correctly, the VEX scheduler stores information on the state of each controlled application thread. When a thread is spawned, it invokes a VEX function, which provides its lightweight-process id to the VEX scheduler. The thread's state is recorded in a data structure that includes the thread's execution status, its current virtual time and current method identifier. A thread can be in one of the following execution states, presented in Fig. 3.3:

- **Running**: Executing on the CPU.
- **Suspended**: Runnable, but waiting to be resumed by the scheduler.
- **Waiting** (indefinitely): Indefinitely waiting/blocked until interrupted or contended lock can be acquired.
- **Timed-Waiting**: Waiting or sleeping for a maximum of a particular amount of time, unless interrupted in the meantime.
- **Performing I/O**: Performing an I/O operation (in the learning or predictive phase as described in Chapter 4).

In order to prevent threads from running ahead in virtual time, only one thread per core is allowed in the Running state (but see Chapter 4).

The states of the controlled threads are placed into a priority queue Q, sorted (in ascending order) according to their virtual timestamps, i.e. the virtual times at which they were last suspended. Among the non-running threads, the one whose execution point is the earliest on the virtual timeline will be on the head of the queue, H_Q. When a thread registers its state, i.e. when it is started, it suspends itself and inserts itself into Q with the current global virtual timestamp.

The VEX scheduler suspends a running thread when the total virtual time since its last resume, exceeds the designated virtual timeslice. When this happens, the current global virtual

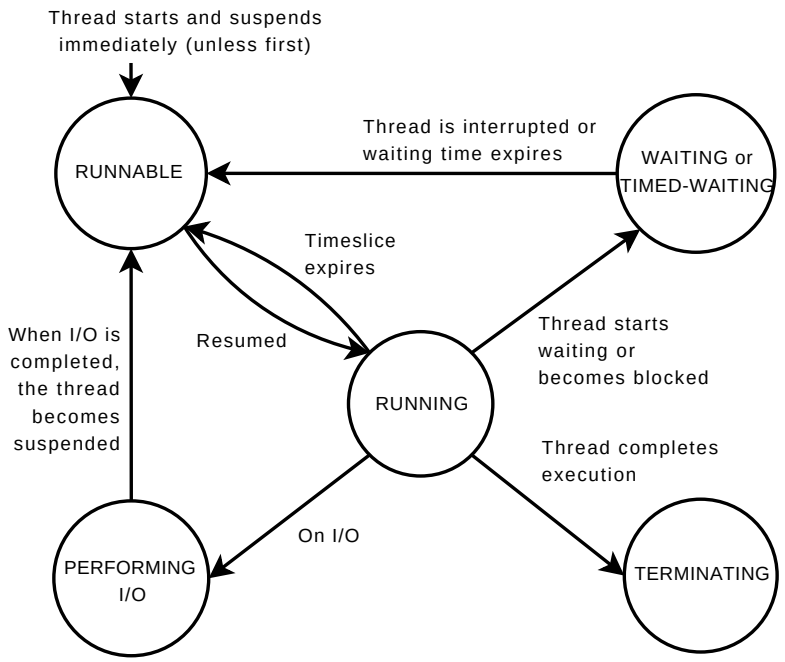


Figure 3.3: Model of VEX thread state transitions

timestamp is updated and the thread is inserted into Q with its current virtual timestamp.

Threads that enter an indefinite wait, or that are blocked waiting for the acquisition of a lock, convey their imminent blocking state to VEX. Once signalled by another thread or acquiring the lock, the waiting thread's state is set to runnable and its local time is updated to the current global virtual time. When a thread sleeps for a specific amount of time t, its sleep time is mapped to *virtual* time, taking account of the scaling factor of the currently executing method, before adding the thread state information to Q. Some care is needed, when a timed-waiting thread is signalled or interrupted by another thread. At that point, the sleeping thread's wake-up time is rolled back to the current global virtual time and its position in Q is adjusted accordingly. When a thread T *yields*, we set its local virtual time to the highest virtual time between all runnable threads plus 1ns, thus allowing all other currently runnable threads to be resumed at least once before T. The yield system call used in the real time execution is not called in the simulation run. To compensate for the removal of the call in the virtual time measurements, VEX profiles the approximate time of the `sched_yield` system call prior to the virtual execution and adds its expected duration to the virtual time of any thread invoking it.

3.1.2 Controlling threads in virtual time

VEX uses the POSIX thread library to control the execution of threads in virtual time, wrapping the operating system scheduler. The VEX scheduler is itself a thread that is spawned before entering the main method of the program. The scheduler allows an application thread to run for one timeslice by invoking a timed wait for the duration of the timeslice, suitably scaled by the *Time Scaling Factor (TSF)* of the thread's currently executed method. For example, if the default scheduler timeslice is 100ms and the current method has a *TSF* equal to 3 then the thread executing it gets a timeslice of 300ms in real time. The method's estimated *virtual time* is then its measured execution time divided by 3, thus adding 100ms of virtual time for 300ms of execution. This simulates the effect of the method executing 3 times faster within a single timeslice.

What happens if the scaling factor changes during the timeslice, for example as a result of a scaled method returning to its calling method that has scaling factor 1? In this case the scheduler is notified and updates the duration of the remaining timeslot accordingly. The scheduler is similarly notified if a method with scaling factor 1 calls a method with $TSF \neq 1$.

The VEX scheduler suspends a thread by sending a POSIX signal to its registered thread-id. A thread T receiving such a signal updates its thread-local CPU time and compares its new timestamp with the thread at the head of the queue. The following scenarios are possible:

- T's virtual time increase is less than a certain percentage p of its assigned timestamp. The OS scheduler has been scheduling other processes (system background processes or the VEX scheduler thread) instead of the running thread selected by VEX, or the running thread has spent most of its time within VEX (whose execution time is disregarded from virtual time measurements as explained in Section 3.2). As our default policy enforces a constant virtual timeslice to all threads, T notifies the VEX scheduler that it will continue execution, while the scheduler sleeps for the remainder of T's virtual timeslice.

- T's virtual time increase is more than p of its assigned timestamp s. In this case, there are two possible courses of action:

- If the virtual timestamp of the head H_Q of the runnable threads queue Q is lower than T's virtual timestamp, then T is suspended, i.e is set to runnable, and remains blocked within the signal handler. H_Q is then resumed.
- If the virtual timestamp of H_Q is higher than T's virtual timestamp, for instance if H_Q is *Timed-Waiting*, then T exits the signal handler and resumes.

In our prototype implementation we have set $p = 75\%$. In principle we could set $p = 100\%$ and request that each thread executes in virtual time no less than its assigned timestamp. We decided against this, to avoid requesting smaller timestamps, when a thread has progressed for a large part of s. For example a thread that would be found to have executed for 90% of s would be requested to run for another 10%. Asking for such small timeslices presents two issues:

- There exist system-related limitations on the lowest duration that can be assigned by VEX as a timeslice to a thread (see following paragraph).
- Assigning larger timestamps allows us to distinguish a thread that has been blocked without VEX knowing it (we elaborate on such "Native Waiting" threads in Chapter 5) and one that has not made enough progress due to OS scheduler decisions.

As the VEX scheduler is itself subject to the OS scheduler, it competes for CPU access with other system processes, that might have an equal or higher priority than it. Even the VEX controlled threads have the same priority as the scheduler thread. As a result, when the timeslice timeout expires, the VEX scheduler thread might not be immediately scheduled, thus increasing the average timeslice assigned to VEX-simulated threads. This creates limitations on the lowest possible timeslices that can be assigned to threads in virtual time and in turn, on the values that can be used as TSFs.

Based on our experimental results presented in Table 3.1, adapted timeslices of less than 1ms in real time for decelerations by a factor $TSF > 100$ (assuming the default scheduler timeslice of 100ms) might not be accurately simulated. The second and third column of Table 3.1 show the

Requested Timeslice [μs]	Normal priority timeslice [μs]		RT priority timeslice [μs]	
	Mean	C.O.V. (%)	Mean	C.O.V. (%)
20	77.05	31.52	24.28	12.63
40	98.48	187.08	44.39	27.02
80	131.14	30.09	84.52	23.18
160	228.11	109.87	164.63	16.43
320	388.56	16.27	324.89	11.18
640	739.88	53.02	644.24	5.33
1000	1079.03	42.46	1003.14	2.29
10000	10604.90	23.00	9953.69	1.19

Table 3.1: VEX actual scheduler timeslices according to requested timeslices: lower requested timeslices are inexact, because the OS schedules other processes after the expiry time. Reported values are for Host-1 (see Appendix A). The means and COVs refer to all samples gathered from a single run with the requested timeslice. Lower timeslices from asynchronous scheduler notification are not taken into account.

mean and coefficient of variance (COV) for the scheduler timeslices acquired for normal priority. The high COVs demonstrate the dependence of each timeslice on the other processes requesting to run at each point. Increasing the scheduler thread's priority in Linux to the highest Real Time (RT), decreases this limit to approximately 20μs (see Table 3.1), thus allowing for decelerations by a factor of 5000. We expect that such values provide an adequately wide range of available *TSF*s to cover meaningful hypothetical virtual specifications. We note here that integration of the VEX and OS scheduler at the kernel-level would further relax these limitations.

3.1.3 Implementation issues

Synchronization issues

The asynchronous signal-based communication, between the VEX scheduler and the application threads constitutes the main implementation challenge. The system shares some data structures between the threads and the scheduler, offering mutual exclusion with locks. If a thread is signaled and suspended whilst holding such a lock then the scheduler will block on that lock, and the system will deadlock. To synchronise between the scheduler and threads, we use a lock, L_T, that is shared between thread T and the scheduler. The idea is that whenever T needs access to a VEX resource shared with other simulated threads, it first acquires control

of L_T. Forcing the scheduler to acquire this same lock, before suspending a thread, overcomes the problem of deadlocks caused by access to shared resources.

A similar issue occurs with heap-management system calls (like `malloc` or `free`), which acquire a kernel-level lock, before allocating or freeing memory. If a thread is interrupted by the VEX scheduler, whilst invoking such a system call and holding this kernel-lock, and then the scheduler tries to invoke a heap-management system call, then the simulation is led to a deadlock. For this reason, it is imperative that the code executed by the scheduler thread, does not invoke any such system call, throughout the simulation (excluding any allocations during VEX data structures initialisation, that occur before the simulation starts). This rule is strictly followed during the development of the framework.

Trapping conditional waiting and signals

Consider the scenario illustrated at the upper part of Figure 3.4. The instrumentation layer invokes the `onThreadWaitingStart` method, before a thread calls `pthread_cond_wait` and the `onThreadWaitingEnd` method, after the thread is signalled to exit the `pthread_cond_wait` call. The purpose of both `onThreadWaitingStart` and `onThreadWaitingEnd` are to convey the state change to VEX. Right before the thread starts waiting, `onThreadWaitingStart` is invoked and VEX changes the thread's state from "Running" to "Waiting". Conversely, after the thread resumes in real time, the thread changes its state from "Waiting" to "Running". In the latter case, however, there exists an inconsistency between the real and VEX thread states, which is illustrated by the shaded part in Figure 3.4. This seemingly low-level detail comprises one of the most important insights in the difference of virtual time execution to discrete-event simulation: in contract to DES where state changes occur exactly at the point where the simulator decides to do so in virtual time, in the virtual time execution it is up to the OS to decide when the thread state changes are conveyed to VEX.

The immediate effect of this is that VEX might falsely consider that all threads are currently blocked in a "Waiting" state, though in fact a thread T_{sig} might have been resumed, but might not yet have updated its state in VEX. Now imagine that a different thread T_{tw} is in the

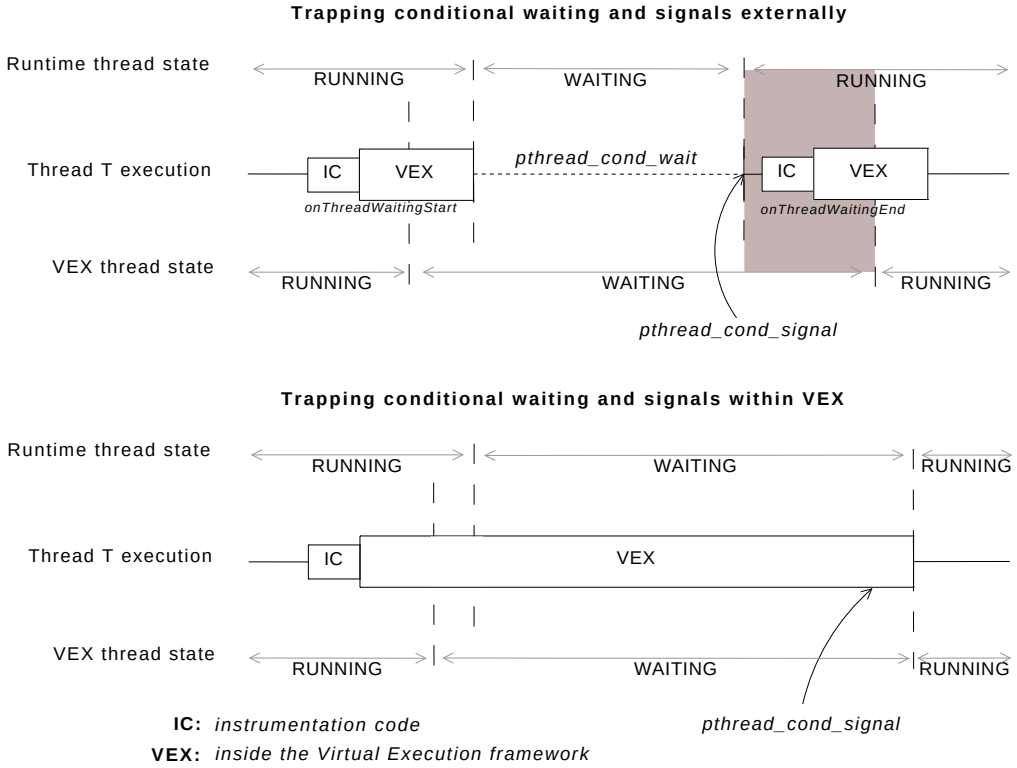


Figure 3.4: Difference between trapping synchronisation events externally and within VEX

"Timed-Waiting" state, waiting for the simulation time to reach its virtual timestamp, which is ahead in virtual time by Δt, with $\Delta t \gg s$ and s the VEX scheduler timeslice. If all threads are falsely identified as blocked, then only T_{tw} is considered runnable and thus the simulator leaps Δt seconds forward in virtual time to the timestamp of T_{tw} and resumes it. When the state change of T_{sig} is eventually acknowledged to VEX, the simulator will assign it a virtual timestamp greater than the one of the recently resumed T_{tw}, thus wrongly increasing all profiled durations of T_{sig} by Δt.

By trapping conditional waits and signals internally, VEX becomes aware of any upcoming state changes, before they occur on the runtime level and thus overcomes this problem. We will be referring to this as the "update-before-event" principle. This is illustrated at the lower part of Figure 3.4. Conditional variables (or monitors) are assigned from the upper layer a unique id, which is passed as a parameter to the VEX waiting/notification related calls. The lock

id is registered into an internal VEX data structure, that stores all threads that are waiting on it. When the conditional variable (monitor) is signaled (notified), one of these waiting threads is selected to be resumed within VEX. Depending on the programming language of the application under study, the original waiting call may still need to be used. For instance, in Java, the `Object.wait()` method would still need to be invoked, to release the monitor of the object. Failing to do so would violate the original program semantics. However, the signaling call is always replaced by the corresponding state altering call within VEX.

Single-core simulation

The scheduling policy of the VEX version presented in this chapter is based on the assumption of a uniprocessor on the simulation target. Only one thread is allowed to execute on the CPU at any time, which enables us to map CPU-times to the global virtual timeline and to model time-scaling by modifying the elapsed time of the only running thread. This simplifies scheduling decisions, synchronisation issues between application threads and VEX scheduler and disregards virtual time coordination problems amongst multiple cores. Nevertheless, resource contention is still exhibited, as thread scheduling decisions and time scaling factors affect the waiting times on locks, the arrivals of notification signals to waiting threads and the queueing of requests to I/O devices. The extensions to VEX to cover multicore architectures are presented in Chapter 7.

Relation to execution-driven simulation

VEX is an execution-based driven simulator, in that it projects execution traces of the application's threads on a virtual timeline. It emulates the functional behaviour of the *virtual codebase*, i.e. the one that would realise the virtual specifications, while simulating the timing behaviour. However, the methodology followed here is quite different from previous work (see Section 2.2.1). The methodology for execution-driven simulation proposed in [32, 40], divides execution into basic blocks, which can be executed independently, i.e. with no interaction between the threads executing them. VEX forces the threads to follow a specific schedule based on the virtual specifications and the scheduling policy selected. This means that as long as

the virtual timeslices are assigned, following a fair-scheduling round-robin scheduling policy, the simulation does not need to take interaction points into account. Only thread state change events need to be trapped into VEX. We note here that the existing scheduling policy of VEX is unaware of thread priorities.

Relation to simulation and emulation

VEX relates to both simulation and emulation. The virtual time execution is based on the idea of using the *TSF*-annotated application code and any method-describing performance models (as we describe in Chapter 8) as a model of the optimised version of the code that corresponds to a "what-if" scenario. VEX is thus a simulator that takes this model as input and simulates it by identifying events that imply a state change and scheduling them according, based on some arbitrary notion of time (i.e. virtual time).

Meanwhile, VEX emulates the OS scheduler of the system that would exhibit the timings determined by the annotated time-scaling factors. This means that the thread scheduling decisions of the VEX scheduler are the same as the ones that would be observed on the system described by these *TSF*s. This is similar to the approaches introduced in the field of network emulation [57, 143] where time-scaling at the virtual machinelevel is used to emulate the behaviour of faster networks or allow for larger-scale experiments with existing hardware [56, 43, 14]. Virtual time execution is more fine grained in that the scaling is applied at the method-level.

3.2 Profiler

The VEX framework is an instrumentation-based profiler that returns the estimated method execution times, as if the virtual specifications applied. We call these measurements *virtual execution times*. If no virtual specifications like time-scaling factors and method performance models are defined, then the VEX profiler is expected to return similar results to a regular profiler. This constitutes our main validation technique throughout this thesis.

The VEX profiler has three objectives:

- to measure and log execution times on a per method level
- to disregard and compensate for the VEX framework's code execution
- to output the results of the virtual execution.

We present these in the following sections.

3.2.1 Measuring virtual time

The framework stores a virtual timestamp counter for each simulated thread. Maintaining up-to-date virtual timestamps for each thread is paramount throughout the VEX simulation, as they determine the scheduling decisions. A *Global Virtual Time (GVT)* counter is also held, a notion borrowed from the Time-Warp parallel distributed event simulation paradigm presented in [70, 69]. The GVT is the minimum committed virtual time amongst all processes, and can only move forward in virtual time.

The virtual timestamp of a thread is initialised to the GVT at the point that the thread was spawned. Every time a thread is resumed it sets its timestamp to the GVT. Threads update their virtual times themselves, when:

- they enter/exit a method
- they invoke any of VEX's calls
- they are suspended by the VEX scheduler.

Virtual timestamps are updated by accessing a thread-local CPU-time counter. CPU-times are increased only during the cycles that a thread is executing in user-mode and disregard overheads from OS scheduling decisions. We note here that though VEX allows only a single running thread per available core, the OS scheduler may decide to schedule any runnable

thread/process (like background system processes). Using elapsed real times would incur an error each time this happened.

Each thread has a current time-scaling factor (TSF), which is originally equal to 1. When a thread enters a method M that has a time-scaling factor $M_{TSF} \neq 1$, then its TSF is multiplied by M_{TSF}. To handle recursive calls correctly and avoid multiplying the TSFs of a method multiple times with itself, each thread keeps a list of all method ids that have affected its time-scaling factor up to this point. Only if the method id of M is not on that list does M_{TSF} change the current TSF. When a method M_x is exited and if the current $TSF \neq 1$, then M_x is found and removed from that list, while the current TSF is divided with M_x's associated time-scaling factor. Time-scaling changes are notified to the VEX scheduler, so that it adjusts its timeslice for the running thread (see Section 3.1.2).

The difference between the current and previous CPU-time is scaled according to the current TSF and then added to the virtual timestamp of the thread. We note here that, due to the decoupled nature of the timing aspect of the VEX simulator, the scaled CPU-time difference is not the only way to model virtual execution times. Alternatively, we could obtain a virtual execution time from a different timing model (based on other system counters, like instructions, cache events etc.), from the simulation of a performance model (as in Chapter 8) or an external simulator. In any case, the thread updates its timestamp locally and commits its virtual timestamp to the GVT, only when it becomes suspended, spawns a child-thread, or terminates execution. This minimises the synchronisation needed between the thread and the VEX scheduler, that might also access the GVT.

The accumulated time resulting from these techniques is accounted for in the virtual timeline according to the model of the underlying hardware. For example, in the uniprocessor scenario, where only one thread executes at a time, virtual times of different threads should be added directly to the global timeline in order to simulate their concurrent (but sequential) execution pattern. In contrast, multi-core architectures (see Chapter 7) imply a level of parallelism between the execution of threads, which needs to be accounted for. In this chapter we focus on uniprocessor architectures.

CPU-times take into account all CPU effects on the user-level, covering performance-affecting factors like branch mispredictions, TLB misses and cache misses. We should clarify here that by changing the thread scheduling during a virtual time execution, we alter the CPU instruction issue sequences and memory access patterns. The performance effects of these changes are now reflected in new CPU-times for all threads executing at that core, regardless of whether they were time-scaled or not. In other words, the virtual time scaling may inflict application wide performance changes beyond the limits of the specified time-scaled methods.

The usage of CPU time does not take into account background system load and also ignores kernel-level time, including system calls and I/O operations. To improve the accuracy of the profiler, we trap these calls and measure their durations in real time (see Chapter 4 for details).

Measuring CPU and real times render the results platform-specific. VEX supports three ways to measure CPU times:

- Invoking the `clock_gettime` system call.
- Patching the kernel to support the *perfctr* module and accessing it directly via our own lightweight library *Perfctr-light*.
- Using the PAPI interface [108]. PAPI can access CPU-time counters, using one of the previous two methods or the *perfmon* module. PAPI is a more extensible solution, because it provides access to other types of counters.

In this respect, VEX offers a performance estimate for the deployment of an application on the simulation host, as described by the CPU time measurements for regular code together with real times for system calls and I/O operations.

3.2.2 Logging method times

The programming-language-specific upper-layer of the virtual time execution framework assigns an id to each profiled method and instruments it to invoke the method entry/exit calls of VEX

with the method's id as an argument. The difference between the thread timestamp upon method entry and upon method exit, is logged as a sample for the estimated execution time of this method. Samples can describe methods or stack traces, providing either flat or call-graph based profiles.

Performance samples are stored on a per-thread basis and merged together when a thread terminates execution. This improves the simulation performance by limiting synchronisation needs, but increases VEX's memory requirements, whilst complicating the exporting of aggregated results at runtime. A solution to the latter that might integrate the VEX profiler into a front-end Graphical User Interface (GUI), to visualise the execution progress is left as future work.

Method entry events are pushed into a thread-local stack and popped when the method exits. Objects describing method entry events are re-used, once their original method is exited as an optimisation in order to avoid memory allocation system calls each time a method is entered.

3.2.3 Compensating for observer effects

VEX is an instrumentation-based profiler (see 2.1.2) and, as such, suffers from *observer effects* [87, 86], which alter the acquired measurements due to the execution overhead of the profiler itself. On top of that, and unlike other profilers, controlled threads may be asynchronously suspended by the profiler at any time, or be blocked waiting to acquire a VEX-related lock. Clearly, delays as a result of executing such code should not be taken into account. Access to the CPU or real time counters also involves an overhead penalty that adds noise to the final results.

The problem is illustrated in Figure 3.5, where a single thread calls method A(), which calls method B(), which calls method C(). The blocks before and after each method describe the instrumentation code (IC) and VEX parts, whose execution times need to be removed. For example the difference between the total inclusive execution time of method A() with these delays (upper line titled as "Total elapsed time") and without the delays (lower line titled

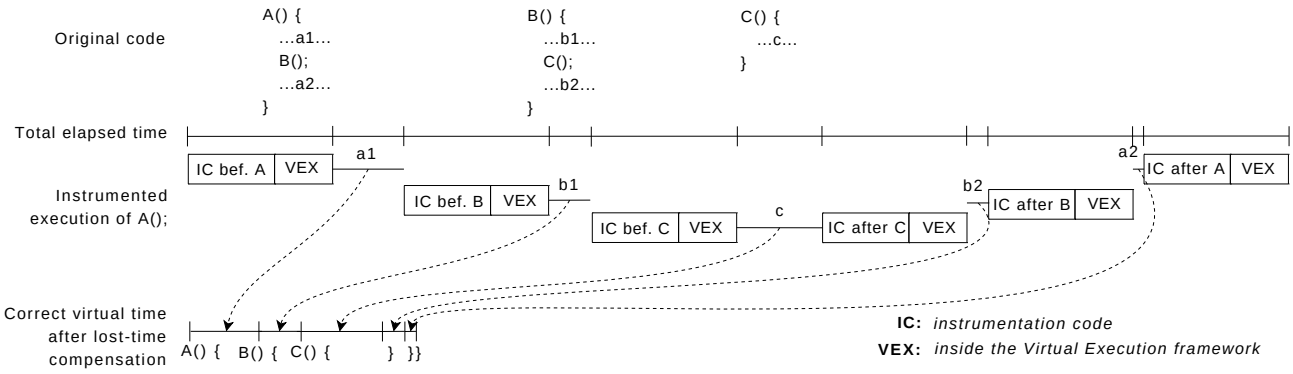


Figure 3.5: Lost time and overhead compensation

as "Correct virtual time...") is significant, though it should be clarified that this depends on the original execution time of the method: the higher the ratio of VEX delay time to original execution time, the larger the error.

Malony et al suggested in [87] to measure the profiler's overhead and compensate for it based on the count of submethods once at the end in the reported results. This approach does not work for VEX, because the intermediate uncompensated measurements would wrongly update the GVT, thus affecting the timestamps of threads that would be resumed next. Specifically, a blocked (waiting) thread would then be resumed at a GVT that would include the profiler's overhead, but without having executed a single method, whilst waiting (which would enable the method count compensation scheme of [87]). To overcome this issue, we follow a scheme based on two time measurements that compensates immediately for VEX and timing overheads, which we refer to as *lost time*.

The key idea to the compensation approach is to remove the lost time immediately upon entry to a VEX-method and to update the virtual timestamp before exiting a VEX-method. In the example of Figure 3.5, this means that the CPU-time counter of the executing thread, is updated at the beginning and end of each "VEX" block. For instance, time is updated:

- on T_{start} at the end of the "VEX" block just before the execution of the *a1* code
- on T_{end} at the beginning of the "VEX" block just after the "IC bef. B" block.

T_{end} - T_{start} provides the execution time of the *a1* code, increased by the duration of the "IC

bef. B" block. VEX measures this duration and subtracts it from T_{end}. It assumes that the time of all instrumentation code ("IC") blocks is the same.

Measuring the duration of an "IC" block is accomplished by repeatedly invoking a sequence of "IC bef."-"VEX" and "IC after"-"VEX" blocks, thus simulating the instrumentation of an empty method, without any arguments and returning value. The empty method's duration should be zero and thus any virtual time measured is a result of noise from lost time. The minimum of all lost time measurements, normalised by the number of iterations, is the estimated time of an "IC" block for the current simulation host and the CPU-timer selection. The result is stored in a file, to be reused for future simulations at the host. The same method is also used for instrumented code around I/O calls.

The advantage is that lost times are not accumulated, leaving the virtual timeline devoid of such overheads. If a thread were signaled to suspend whilst executing a VEX method, then the virtual timestamp update in the signal handler would include additional time from the execution of the VEX code. Therefore, a shared key between scheduler and each thread ensures that the thread is not blocked when inside VEX code. Note that we do not use spin locks, in order to avoid affecting the CPU-time counters of a possibly waiting thread.

Compensating for lost time is one approach to limiting observer effects. However, it is rather simplistic as it does not deal with indirect effects to cache and memory alignment, caused by the execution of VEX code. In particular, for methods whose original execution time is lower or equal to the VEX code execution and whose total execution times are relatively low (e.g. setter and getter methods), the compensation might yield inaccurate results. Our experiments (see Section 3.3.2) agree with Malony et al, that remark in [87] that "A few nanoseconds in overhead estimation are enough to cause major compensation errors". Therefore, in such cases it is preferable to remove the instrumentation altogether. This is accomplished with the *adaptive profiling* scheme.

The adaptive profiling scheme comprises an *invalidation policy* at VEX level and a corresponding support module that enforces the policy at the upper-instrumentation layer (see Section 5.1.3). The invalidation policy determines whether a method should be profiled or not.

Three criteria can be used for invalidating methods:

- The total number of method invocations is greater than N.
- The mean execution time of the method is smaller than a defined value t.
- The total execution time of the method is less than a defined percentage p of the total execution time.

The decision on whether the instrumentation code of a method needs to be removed according to the invalidation policy or not is then forwarded to the upper-instrumentation layer. In turn, the instrumentation layer takes the necessary actions to stop profiling the method, thus limiting VEX's observer effects.

3.2.4 Simulation output

Profile

In the end of a simulation run VEX returns method-level execution time predictions, based on the virtual time execution. Results are presented either on a per-method stack trace or per-method basis, depending on whether the user requested a *path* or a *flat* profile. VEX results include the following measurements for each method or stack trace:

- Number of invocations.
- Mean predicted CPU-time excluding called methods (ExCpu).
- Mean predicted total execution time excluding called methods (ExTotal).
- Mean predicted CPU-time including called methods (InclCpu).
- Mean predicted total execution time including called methods (InclTotal).
- Percentage of predicted inclusive total time (InclTotal) on total execution time.

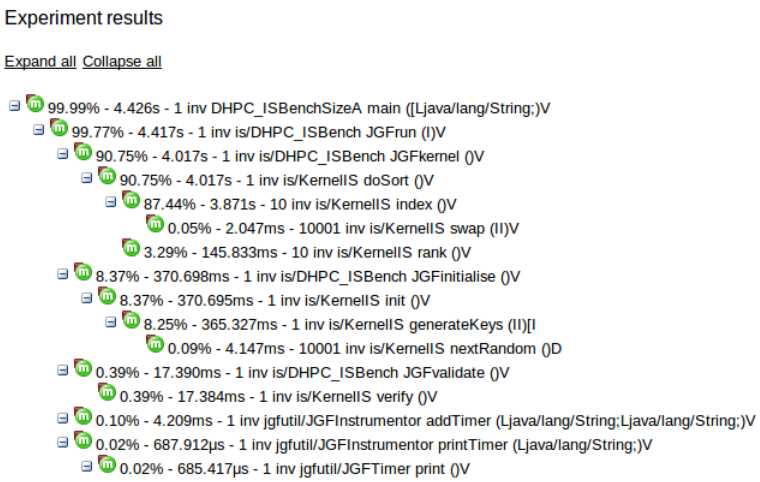


Figure 3.6: Per stack-trace profile visualisation

CPU-times disregard those periods where a thread is waiting or runnable. Results are printed in a Comma-Separated-Value (CSV) file that is easily processed by spreadsheet management applications, enabling further processing or graphical representation.

Method-level profiles can be exported in an *hprof*-like style, returning only execution time percentages including and excluding called methods. This allows for direct comparison with standard profilers.

Results on a stack-trace basis are also presented graphically (Figure 3.6), with a simple html-based interface, that expands submethods in a tree-like way. Users can define a *prune_percent* threshold, to prune the branches of the tree that have a total estimated time less than this threshold.

Virtual times

Apart from a profiler, VEX is a virtual execution runtime environment. This enables a different usage of the framework that is based on the upper instrumentation layer replacing calls that return time on the application-level, with VEX calls returning virtual timestamps. For example, all invocations to the `gettimeofday` call are replaced to return the global virtual time at that point. This means that other runtime tools can be executed in virtual time, allowing performance metrics, like response times or throughput to be replaced by their virtual counter-

parts. This option is on by default: if it is disabled then real times can be returned as normal, to measure for instance the overhead of the framework. Most of our validation tests keep the default value on and measure overhead with timers external to the VEX simulation.

Virtual execution visualisation

The VEX visualiser is a component that offers a graphical representation of the virtual execution. It gathers event samples throughout the simulation and uses them after it terminates to show the execution progress. Originally used as a debugging and validation tool, the visualiser can be used for performance analysis or educational purposes.

The event log is accessed by a Java applet, which allows users to follow the progress of the virtual execution step by step, by clicking to show the next event. Users can jump forward to any time point of the virtual execution, or zoom in to particular time intervals.

Contrary to method time logging, the visualisation component stores events in a centralised way, which increases the overhead of the framework. The analysis capabilities of the visualiser are, in practice, limited to a few tens of threads. This allows the virtual timeline to fit in the same window as the simulated threads. Events are gathered throughout the virtual execution and are only stored and used after it finishes, which increases the memory requirements of the simulation as well. For these reasons, the visualiser is useful for short, validation or experimental runs.

In Figure 3.7 we show a virtual execution with four threads. In this program, thread *main* creates three threads, that execute exactly the same code, which performs a fixed number of loop calculations. In the virtual execution, the method executed by thread:

- *SlowThread* is assigned a *TSF* of 0.2, resulting in a 5 times slowdown
- *FastThread* is assigned a *TSF* of 5.0, resulting in a 5 times speedup
- *NormalThread* is left unaltered.

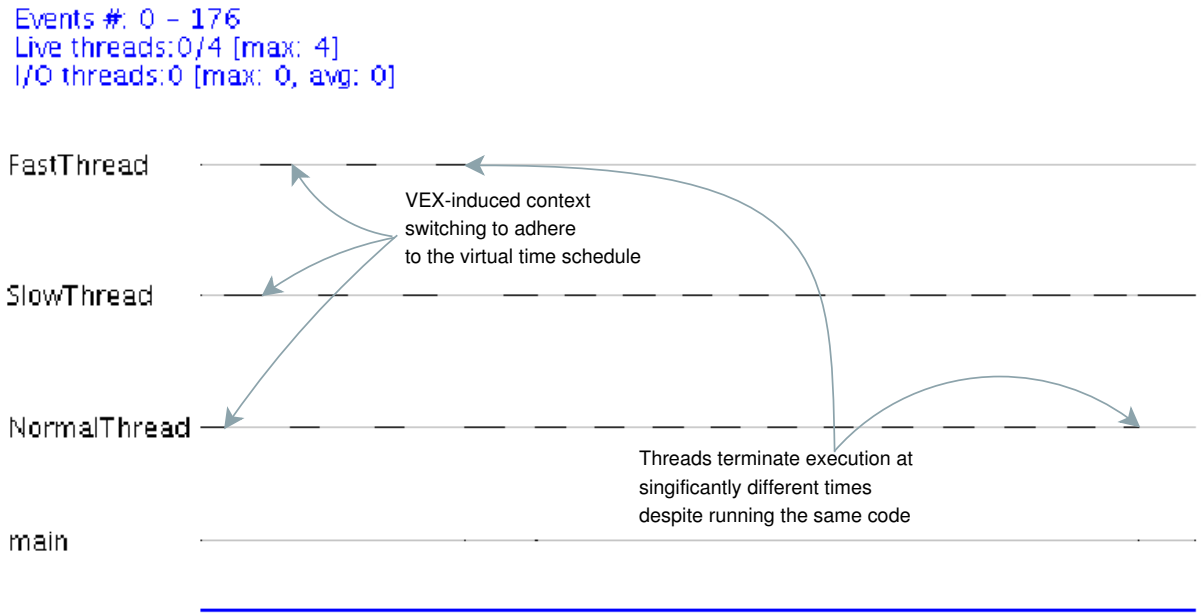


Figure 3.7: Visualiser snapshot: all threads apart from main execute the same code with different virtual specifications and progress accordingly on a uniprocessor machine. The dashes signify timeslices that a thread is in the "Running" state.

The visualiser shows how VEX simulates the execution of the three threads on a uniprocessor correctly, assigning timeslices in a round-robin way. The blue line on the bottom of Figure 3.7 is the virtual timeline.

Statistics

VEX's results include statistics for the virtual execution. By default the framework generates a transition graph for the VEX states, a feature which summarises the simulation characteristics. Especially for larger-scale experiments, where the visualiser is unsuitable, these kinds of statistics provide a useful insight into the virtual execution.

Figure 3.8 illustrates such a graph for the execution of 32 threads spawned by *main* (in total 33 registering and terminating threads). "Suspended" threads became "Running" 292 times and either remained in that state at the end of their timeslice (in 34 cases) or became suspended again in 257 cases. Thread *main* enters the "Waiting" state, when trying to join with a child-thread that is still running. The state transition graphs help us to identify how specific state

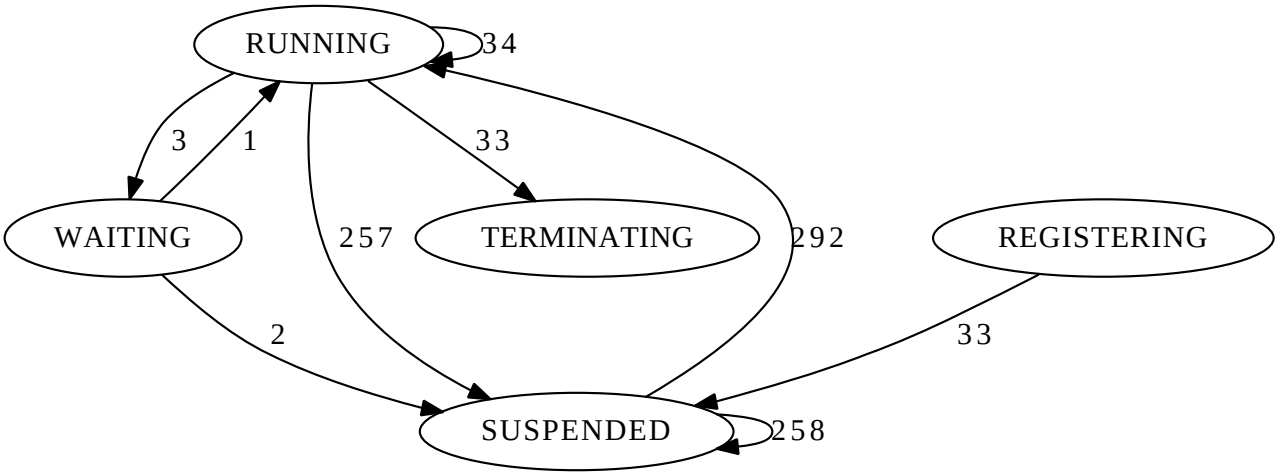


Figure 3.8: A sample automatically generated state transition graph

changes affect the simulation results and execution time. They have been used in the evaluation of VEX.

VEX also provides analytic time-related statistics, showing the virtual time progress. This feature is user-enabled and may gather results on a global scale, on a per-thread basis or separately for each new time sample created. Using time statistics we are able to compare real and virtual executions. To facilitate this comparison, VEX offers a "pure profiling" version, in which the scheduling part is entirely disabled. We call this *no-scheduling* VEX. Method entries and exits are still trapped and statistics generated, but the framework does not act upon the various thread state changes, nor does it assign timeslices or control the scheduling of threads.

The framework may optionally export scheduler-related statistics, namely timeslices assigned, interrupts sent and virtual time progress as a result of this scheduler's activities (used in multicore simulations).

Another set of statistics is offered with the help of system and PAPI counters. These range from instruction counts and L2 misses to thread signals, context switches, memory requirements etc. We compare these measurements between the real-time and virtual executions, to understand quantitatively how the VEX simulation affects the performance of the application.

3.3 Evaluation

In this section we present the experiments that we have performed in order to validate that VEX implements the intended design and to identify its overhead and limitations. We invoke the functionalities of the core by *manually* instrumenting the corresponding code of small C++ test programs. This process would normally be performed automatically by the upper language-specific layer, but this approach removes the complexities of higher-level layers and allows us to isolate the behaviour of the VEX core. All following results are acquired on the uniprocessor configuration of Host-1 (see Appendix A).

3.3.1 Validation

Scheduling sequence

The first claim we validate is the extent to which the VEX user-level scheduler bypasses the decisions of the OS one. Our testing program spawns threads in groups of three: a "normal" one, an "accelerated" one and a "decelerated" one. All threads from all groups execute exactly the same method, `calculateE()`, which calculates the Euler number e. The only difference is that the calling methods of `calculateE()` for "accelerated" and "decelerated" threads are assigned a time-scaling factor of 2.0 and 0.5 respectively.

We illustrate the effect of VEX by executing this program first normally in real time and then with VEX. In each run we create 5 triplets of threads in a round-robin fashion. In the real time execution, we get a random ordering of the threads, as the OS scheduler treats them all in the same way. In the virtual time execution, all 5 accelerated threads, finish before the 5 normal threads, which finish before the 5 decelerated threads. Increasing the default 100ms timeslice of the virtual execution to 1sec gives enough time for non-accelerated threads to finish within the first timeslice they are assigned. This perturbs the strict ordering, though this effect is expected and is overcome if the workload of each thread is increased.

By enabling the VEX visualiser and zooming in on an interval in which none of the threads

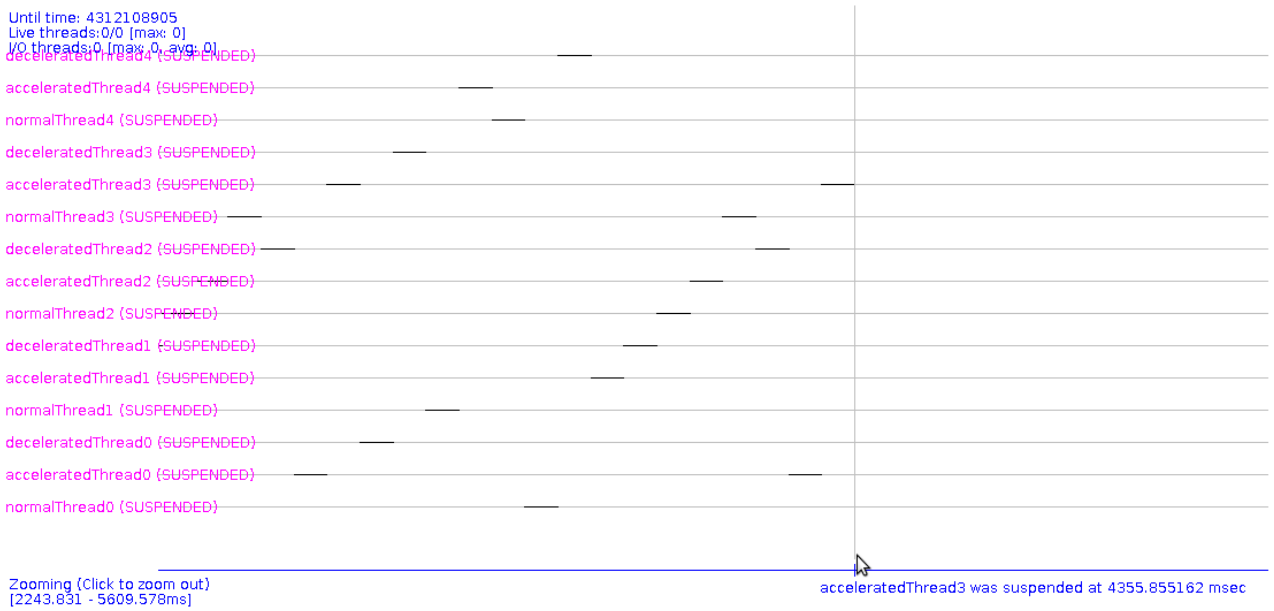


Figure 3.9: Zoomed in visualisation of the scheduling sequence test as output by the VEX visualiser

has terminated, we acquire the virtual execution depicted in Figure 3.9. Only one thread is progressing at any point on the virtual timeline, as we have defined a uniprocessor simulation. Another interesting observation is that, despite their type, all threads are assigned the same virtual timeslice. However, this corresponds to a different amount of real time, according to the defined method time-scaling factor. We note here that by simply adjusting the scheduling priorities of the threads, we would not be able to simulate this kind of execution.

Profile accuracy

One of the main research challenges of VEX concerns the accuracy of the profile, considering the highly-intrusive nature of instrumentation-based profiling and the fact that VEX controls the scheduling of the application threads. Ideally, VEX predictions on a complete codebase without any virtual specifications should agree with real time measurements on a regular execution.

We investigate this issue by generating a random program and executing it once in real time and once in virtual time, without any virtual specifications. The random program is composed of methods that invoke a random number of other methods or perform a short calculation (see Figure 3.10). By setting the seed of the random number generator to be the number of

```
void MethodSimulation::runMethod() {
    VEX::methodEntry(methodId);
    int submethodsCount = 0;
    int operationsCount = 0;
    for (int i = 0; i < totalSize; ++i) {
        if (nextActionIsSubMethod[i]) {
            (subMethods[submethodsCount++])->runMethod();
        } else {
            (methodOperations[operationsCount++])->calculate();
        }
    }
    VEX::methodExit(methodId);
}
```

Figure 3.10: C++ presentation of the random program executioner. The next element of execution is either a sub-method or a loop-based calculation

requested methods M, we always create the same random program for each M. This logic is followed recursively for each of a specified number of methods. The main motivation for using a random program is that we can set the random generator to insert method entry/exit instruments in each created method and so simulate a virtual execution in a scalable way. Running an existing computation kernel, would require us to manually instrument the points where threads change their states. We postpone such tests for the integrated version with the automatic instrumentation layer.

The results of Figure 3.11 show the virtual execution accuracy for various random programs, differentiated by their total method count M. By accuracy we mean how close the virtual execution estimated time is to the real execution total elapsed time. We note here that despite the fact that some confidence intervals span positive values (i.e. show the prediction being higher than the observed real time), in fact all predictions are smaller than the observed times. This is because the simulator uses CPU-time (see Section 3.2.1) and so does not take into account background system effects like other processes being scheduled, or the execution time of the OS scheduler itself. The smaller the total execution time of the test the higher the variability in the observed real times, which leads to the wide confidence intervals of the prediction accuracy for up to 16 threads.

[1]We note here that throughout this thesis we assume that all measurements are normally distributed and use the Student's t-distribution to define confidence intervals.

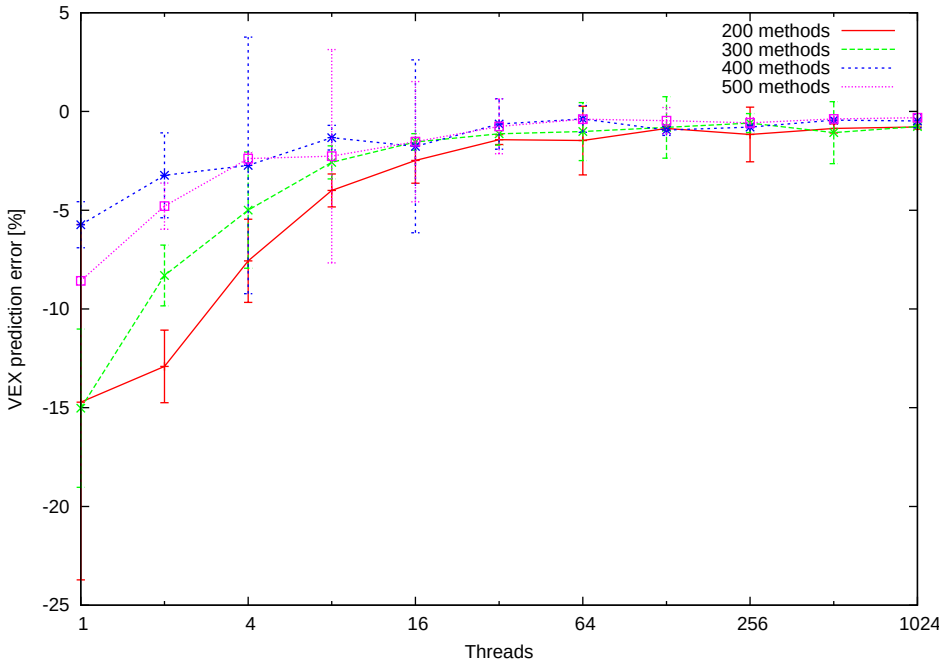


Figure 3.11: Average prediction error of 10 measurements on Host-1 for 1 to 1024 threads and 200 to 500 random methods with approximate 95% confidence intervals[1].

3.3.2 Overhead

The main source of VEX overhead is the method entry and exit event logging. During each such method entry, the CPU-time counter is accessed twice (for lost time compensation purposes), the virtual time of the thread is updated and a timestamp is generated and pushed into the thread-local profiler stack for this event. This generated event is popped at method exit. I/O methods have a higher mean load, as a result of registering an identification point for each distinct I/O invocation point (presented in Chapter 4).

Table 3.2 shows the mean times in nanoseconds for various CPU-time counter selections. The

Timer	**Method instr. [ns]**	**I/O method instr. [ns]**
Perfctr-light	408	2294
PAPI+perfctr	519	2311
PAPI+clock_thread_cputime_id	524	2308
PAPI+times	532	2357
PAPI+proc	737	3322
PAPI+getrusage	794	3668
Clock_gettime	4915	10504

Table 3.2: Method instrumentation overheads for different CPU-timer selections

fastest “Perfctr-light” counter requires the patching of the Linux kernel with the “Perfctr”

module. Although the "PAPI+Perfctr" setup also accesses CPU-time counters through the "Perfctr" module, our "Perfctr-light" timer setup outperforms it by approximately 20%, by caching the pointer to the performance counter within VEX on a thread-local variable and removing some PAPI-related checks within the CPU-time returning code.

Overhead compensation

Compensating for delays incurred by VEX is essential for increasing the performance prediction accuracy and maintaining a correct interleaving of the simulated threads. In this section, we demonstrate the effect of the compensation scheme on VEX predictions. The instrumentation overheads are profiled as described in Section 3.2.3, namely by measuring the virtual time wrongly accounted for an empty method. We also show how small differences in code execution can affect the compensation success.

The testing program executes a loop that calls a method M in each iteration. We select three short computation-based methods for M, so that the ratio of VEX instrumentation code time to the execution time of M is large. The three resulting programs are executed in real time first and then in virtual time without and with compensation using the measured instrumentation code delays.

Test - g++Flag	**Real time**		**VEX Prediction**		**Prediction error (%)**
	Mean [s]	**C.O.V. (%)**	**Mean [s]**	**C.O.V. (%)**	
Loops1-O3	0.121	1.14	0.893	5.72	639.32
Loops2-O3	0.069	3.78	0.901	4.04	1200.81
Loops3-O3	0.429	1.10	1.312	2.13	205.79
Loops1-O0	0.202	5.21	0.908	3.55	349.12
Loops2-O0	0.181	9.83	0.966	4.02	432.59
Loops3-O0	0.507	2.86	1.353	3.31	166.81

Table 3.3: Prediction errors for trivial short methods without any overhead compensation. Results from 100 measurements on Host-1 are shown

To investigate the behaviour of the profiler under such circumstances, we define three different methods that perform a short calculation (1-3 lines of code each) and which are invoked iteratively within a loop (called Loops1, Loops2 and Loops3). All method invocations are manually instrumented to call VEX method entry and exit calls. Setting up the experiment in

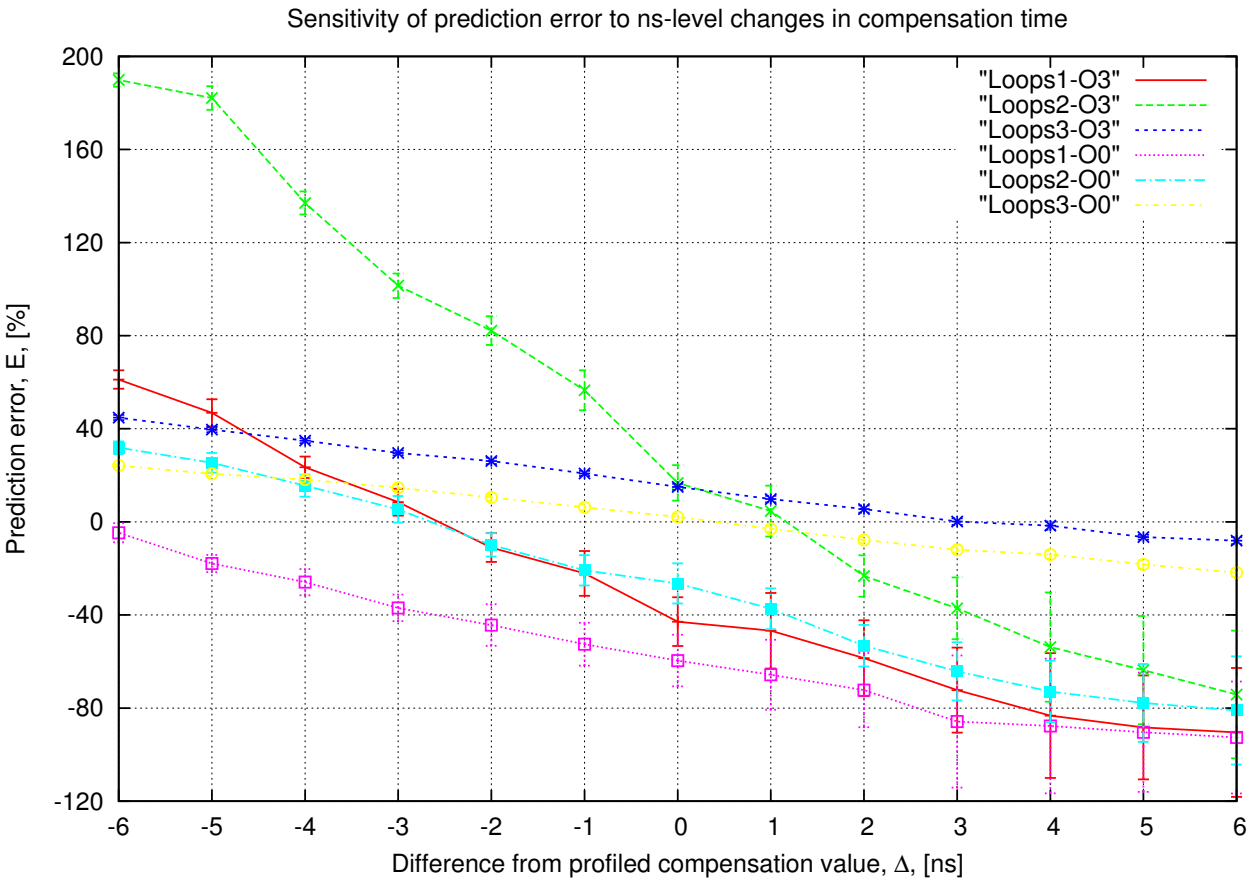


Figure 3.12: Demonstrating the effect of a single nanosecond in the lost time compensation scheme in the prediction error of VEX for three trivial short methods and two different compilation options. Results are from 100 measurements on Host-1 and 95% confidence intervals.

this way, we expect the observer effects from VEX profiling to be high. Indeed, without any compensation we get the predictions errors of Table 3.3, which are on average 5 times higher in comparison to the real time execution.

In Figure 3.12 we demonstrate how sensitive the compensation mechanism is when methods that are "short" in relation to the VEX overhead are invoked at a high rate. As explained in Section 3.2.3, VEX compensates for the additional execution time caused by the invocation of instrumentation code by profiling the corresponding code and acquiring a compensation value C. C is measured once for a simulation host and is then retrieved (from a file) for all virtual time executions on that host. Using C for overhead compensation in each of our test programs, we acquire the prediction errors that correspond to $\Delta = 0$ on the x-axis of Figure 3.12. The results for this compensation value vary from a relative overprediction of 20% for "Loops2-O3" to an underprediction of 60% for "Loops1-O0". As we change C by 1ns, we find that each

program yields the best prediction results ($E = 0$ on the y-axis) for a simulation with a different compensation value. For example, "Loops1-O3" and "Loops2-O0" return the lowest prediction errors for $-3 < \Delta < -2$, "Loops3-O0" for $\Delta = 0$ and "Loops3-O3" for $\Delta = 3$. The sensitivity of the prediction error to a change of 1ns (observed by the slope of the lines corresponding to each test) also varies amongst the different test codes.

This could be attributed to the fact that the code of each test is optimised differently by the compiler (because of the use of the "-O3" g++ flag), thus generating different binaries in each case and affecting the code before and after the invocation of the VEX instruments. Disabling the optimisation and disassembling the code, we find that the additional code is exactly the same in all instrumented methods. The results for this setup are presented in Figure 3.12 for the "Loops*-O0" measurements show that the behaviour is similarly erratic, leading us to believe that the effect of the invocation of the instrumentation code to the CPU pipeline, instruction execution order and memory access patterns is different in each case and can only be approximately compensated. We conclude that the effect of the compensation scheme is beneficial to the accuracy of the prediction, but that it does not provide a precise solution: in extreme cases like the ones presented here it is preferable to remove the instrumentation altogether. This is the responsibility of the upper instrumentation layer presented in Section 5.1.3.

3.3.3 Limitations

Page faults

VEX uses CPU-time to measure the execution time of user-level code and real elapsed time for system calls and in particular I/O calls. This approach covers the execution times incurred by most program behaviours; however, it does not deal with major page faults. In Linux, these happen when a page is not loaded in memory and needs to be fetched from the disk. This leads to a kernel trap, which executes the page fault handler to load the page. This time is not measured by the CPU-time counter and, without supporting the VEX simulation at a lower level, we can not know when the page faults occur to measure their duration in real time.

Pages [k]	**Minor Faults**	**Major Faults**	**I/O blks**	**Real time[s]**	**VEX time[s]**	**Err%**
64	262,672	0	0	0.79	0.63	-20.60
128	525,328	0	0	1.50	1.24	-17.06
192	795,769	565	39184	26.72	2.08	-92.22
256	1,058,422	609	42280	58.66	3.30	-94.37
320	1,321,090	642	43604	100.85	4.70	-95.34
384	1,583,735	679	45156	203.72	15.03	-92.62

Table 3.4: Major page fault effect on VEX prediction accuracy

We demonstrate this problem by executing a program that loads arrays of bytes equal to the system page size (4KB for Host-1 of Appendix A) and touches random bytes in random pages. As we increase the requested pages, the number of page faults increases and the performance prediction of VEX becomes less accurate. This is shown in Table 3.4.

Although this limits the applicability of VEX in identifying page thrashing effects, measuring the number of major page faults of a simulated process could indicate the extent to which VEX predictions are expected to be accurate. Besides that, instrumenting the page fault handler to measure its execution and map the corresponding time onto the virtual timeline, is possible for a kernel-level VEX implementation. We expand on both these ideas for future work in Chapter 9.

Chapter 4

I/O in virtual time

The I/O subsystem remains today the least efficient part of a computer system, with a wide gap between the performance of I/O and CPU operations: accesses to disks or network requests are millions of time slower than CPU cycles. In order to be used as an efficient profiling and hypothetical scenarios investigation methodology, the virtual time simulation needs to include I/O durations. To generate a "realistic" thread-interleaving pattern, the virtual time execution should simulate Direct Memory Access (DMA) I/O operations [131]. By allowing for other threads to obtain CPU control, while a thread is issuing an I/O and waiting for its results, DMA approaches support parallelism between CPU execution and I/O. If any of the threads that execute on the CPU issue further I/O requests, then the load on the I/O subsystem is increased.

We employ an I/O prediction scheme to correctly simulate DMA behaviour, enable multiple threads to issue I/O requests in parallel and allow time-scaling of I/O operations. When an I/O invocation point is encountered, the simulator estimates the expected duration T_{pred} of the request and predicts the virtual time PVT when the operation will complete as $PVT = GVT + T_{pred}$, where GVT the current global virtual time. Other threads are then allowed to make progress (executing on the CPU or issuing new I/O requests) in parallel with the original I/O request, as long as their virtual timestamp is less than PVT. Threads with virtual timestamps higher than PVT are stalled until the I/O request completes. This approach also

generates a *correct virtual schedule* (see the introduction of Chapter 3 for the definition) in light of I/O time-scaling (with time-scaling factor TSF): only the threads that have virtual timestamps less than $PVT' = GVT + T_{pred} \cdot TSF$ will be executed on the CPU in parallel with the I/O operation. This simulates the load that would be observed on the I/O subsystem, if the operation's duration was indeed scaled.

The reader should note the distinction between the predicted I/O completion time PVT and the observed one OVT (with I/O duration T_{obs}). Theoretically, it is the OVT that should determine which threads are allowed to progress in parallel with the I/O operation. Only because OVT is not known when the I/O system call is invoked, do we estimate it with PVT. The closer the predicted I/O completion time PVT to the observed one OVT, the more consistent the prediction-based thread schedule to the one that *would be* generated in the *correct virtual schedule*. Special handling is needed to compensate for over- or underprediction of OVT by PVT. We note here that our attempts on reusing previous samples to acquire good predictions of the imminent I/O load in virtual time are quite disparate from the field of I/O trace replaying (see Joukov et al in [72]). In that case I/O traces are captured in a trace run to enable the exact replication of the I/O load in a future one, an approach which is useful for stress testing and debugging activities. In contrast, replaying in the context of VEX reuses previously observed I/O durations to predict the next ones and create a correct virtual time schedule. VEX thus only determines the threads that are allowed to run in parallel with I/O operations: the execution of the simulated program is the one that eventually determines the exact point when an I/O request is issued.

Our prototype implementation of this methodology in VEX supports a number of options to allow its users to configure how PVT is going to be estimated in each simulation run. Parameters include the level at which previous measurements of I/O operations are grouped together to offer a prediction, the number of samples that are needed and the way they are used. VEX offers several different prediction methods, which can be categorised in the sliding-window replay, the basic statistical, the distribution approximation and time-series ones. Although similar to other black-box methods [3, 74, 138] in terms of input parameter selection, our approach differs in that the prediction itself affects the progress of the simulation and consequently the future

measurements on the I/O subsystem. The users may select at simulation startup which method will be used. Users can also set an I/O threshold I_T, to distinguish between the I/O requests that are serviced by the page cache and the ones that reach the I/O device. When the prediction for the duration of an I/O request is higher than I_T, VEX simulates it as if it performs an I/O request to the device and allows other threads to progress in parallel. Otherwise, it measures the duration of the system call without context switching the I/O issuing thread. In addition, VEX allows users to analyse the effect of I/O to the total estimated virtual time, by offering an I/O-exclusion parameter, which determines the percentage of random requests whose duration should not be taken into account.

The main contribution of this chapter is the methodology for performing I/O in a virtual time execution and the prototype implementation of the approach in VEX. Using benchmarks that stress the I/O-subsystem, we evaluate the various I/O prediction methods and parameters in an attempt to identify an optimal configuration. We conclude that though no consistently optimal configuration is found, prediction methods that favour overpredictions ($PVT > OVT$) instead of underpredictions, or that make use of an (even crudely approximated) I/O threshold, yield better results on the total predicted time of the benchmark. We also experiment with time-scaling and I/O-exclusion parameters, showing how they affect the virtual time.

4.1 I/O in VEX

4.1.1 Requirements

Profiling and scheduling in the virtual time execution framework are based on CPU-time counters, which measure execution times of user-level code. This renders virtual time scheduling independent from the OS: regardless of the latter's scheduler decisions, only the actual running time of a thread is measured. As cycles in kernel mode, i.e. system calls including I/O requests, are not accounted for by CPU-time counters, the virtual time simulator must be notified of their entry and exit points and measure their duration in wall clock time. This in turn affects

the reproducibility of the measurements, because the real time duration depends on when the OS scheduler reschedules the I/O requesting thread to allow it to stop the timer.

An obvious approach would be to issue each I/O request separately, measure its duration in real time and then map it on the virtual timeline, overlapping with other I/O requests if needed. Not only would this add to the total simulation time, but it would incur inconsistencies in the way virtual executions are simulated. Specifically, if under a hypothetical scenario all I/O operations were to take place together, contending for the same resources, their expected time would be different to the one provided by the real time measurement of a single thread performing a single a I/O. Concurrently issuing I/O requests also probes the performance impact of the system's I/O merging and scheduling policies. These effects are disregarded if each I/O operation is issued in isolation.

A third issue concerns DMA I/O operations to disks, that force an I/O-waiting thread to yield the processor to a runnable one. This generates parallelism between I/O and CPU operations or between multiple I/O operations even on a single-CPU platform. The challenge here is to ensure that I/O contention and I/O parallelism are also reflected in the virtual time schedule.

4.1.2 Methodology

All system calls need to be profiled in wallclock time and thus be trapped into our simulation framework. For the purposes of virtual time execution, we can distinguish the trapped system calls in three categories:

- **Regular**: any process or memory handling system calls.
- **Cached I/O**: invocations of system calls like `read` that are serviced by the page cache and should not yield the processor to another thread.
- **Standard I/O**: any invocations of system calls like `read` that perform an I/O request and should allow for other threads to progress in parallel.

These three categories are discussed here in order.

Regular system calls

Regular system calls do not allow other threads to progress in parallel. The only required handling is to measure the duration of the system call in wallclock time and update the virtual timestamp of its calling thread accordingly. It then resumes the simulation of the currently running thread as normal. The distinction between regular system calls and any I/O-related ones is the responsibility of the upper instrumentation layer. For example, a method id could be assigned to each system call and the ids of regular system calls would belong to a particular set. In this chapter we do not take into account regular system calls, but focus on the I/O ones that comprise a more challenging research problem.

I/O system calls

The "Cached I/O" and "Standard I/O" categories are collectively referred to as I/O system calls. The main objective here is to identify and handle each type differently. Specifically, all I/O system calls are simulated using a learning and a prediction period. During the initial learning period, threads issue I/O operations as normal and the observed I/O times are accounted for in real time. Other threads are always allowed to execute in parallel during the learning period. The observed I/O durations of this period are stored in internal data structures, that are sampled in the prediction phase. Storage approaches and sampling depends on the prediction method used (see Section 4.2). The number of samples that are needed before entering the prediction phase is also configurable.

When an I/O system call is encountered at the prediction phase, then a new prediction T_{pred} for its duration is made. This determines whether this system call belongs to the "Standard I/O" or the "Cached I/O" sub-category as follows:

- If no I/O threshold is defined, then all I/O operations are treated as "Standard I/O" requests.
- If an I/O threshold I_T is defined, then:

- if $T_{pred} < I_T$, then the I/O operation belongs to the "Cached I/O" category
- if $T_{pred} \geq I_T$, then the I/O operation belongs to the "Standard I/O" one.

In this respect, the prediction is used to determine whether the request exceeds a "fast/slow" threshold, which is also found in [74].

We elaborate on these two categories in the following sections.

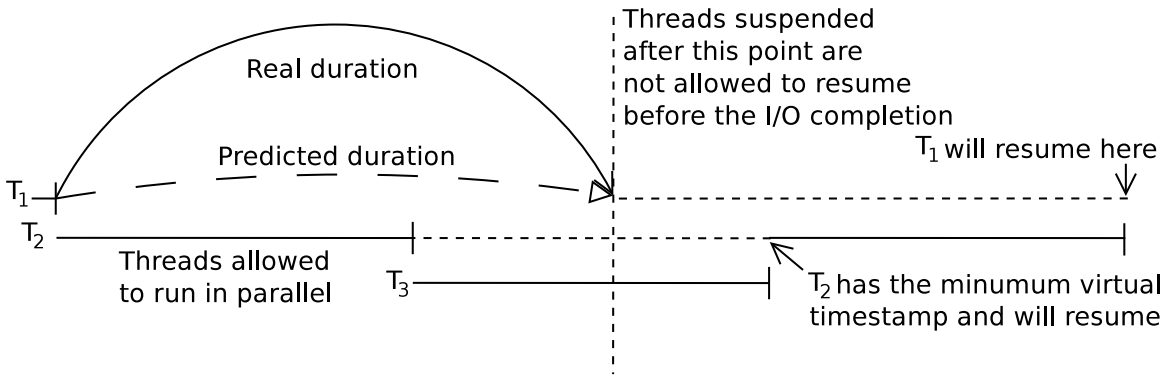
Cached I/O system calls

Cached I/O operations are treated as "Regular" system calls, having their execution time measured in real time, but without yielding the CPU to another thread. As I/O requests still are at least one order of magnitude slower than memory accesses [55], we show in our evaluation study how we can determine such an I/O threshold based on a histogram of previous I/O request durations.

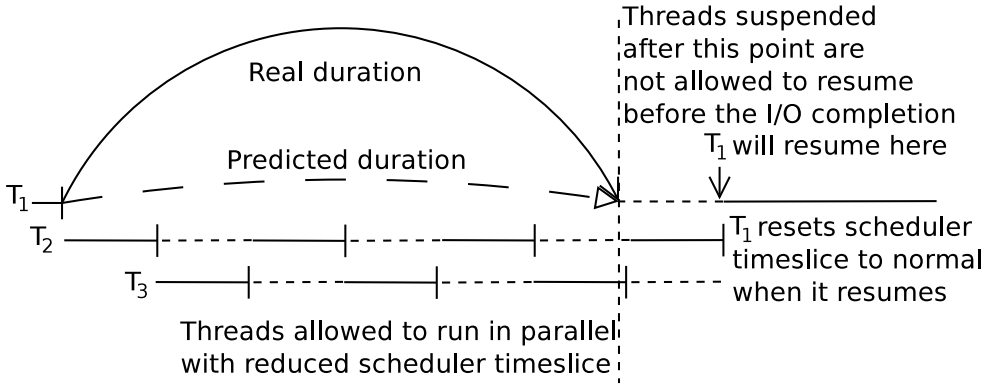
Standard I/O system calls

In the case of a standard I/O request, the virtual time of the calling thread is increased by T_{pred} to PVT. Then the simulation resumes the next thread on the virtual timeline, but *without suspending the former*. This will only happen upon completion of the I/O operation at OVT. At that point the virtual timestamp of the thread is updated according to the observed real time of the I/O operation and the thread is pushed into the queue of runnable threads to be scheduled again according to its new timestamp OVT. If at any point the predicted I/O completion time of a thread is the minimum amongst all threads, then the simulator waits for it to complete. Otherwise, runnable threads that are currently suspended at a virtual timestamp lower than PVT are allowed to resume before the I/O completes. This is illustrated in Figure 4.1a: T_2 and T_3 are resumed, before T_1 completes its I/O operation.

An interesting issue here is that if N threads are simulated in total, with one of them, T_1, issuing an I/O request, then up to $N-1$ threads can resume before T_1 does so after completing



(a) Normal timeslice



(b) Minimised timeslice

Figure 4.1: I/O prediction methodology and effect of reducing the virtual time scheduler timeslice to simulate priority boosting

the request. This can lead to a delay of at most $(N-1)s$ until T_1 is rescheduled, where s the scheduler timeslice. If s is higher than the average I/O duration (typically true for $s = 100ms$), then this delay limits the significance of the prediction scheme altogether. This also contradicts the Linux priority boosting approach, where threads that have just completed I/O are rescheduled sooner than other runnable threads. To overcome this and properly simulate priority boosting, we impose a temporary reduction of the scheduler timeslice to s_{min}. This allows the virtual time scheduler to monitor the progress of threads at a higher rate, thus decreasing the maximum waiting time of T_1 after the completion of the I/O operation, to $(N-1)s_{min}$ (see Figure 4.1b). This approach also limits the effects of I/O duration misprediction (see Section 4.1.3).

By repeating this process for more than one thread, we simulate the effect of issuing I/O operations in parallel and of stressing the underlying I/O subsystem accordingly. This offers a simulation of I/O parallelism that is consistent to the one that would appear in a regular

execution. Parallel I/O operations are also profiled in real time and the values measured are grouped together in buffers, according to the number of simultaneously operating threads N. If a thread requests an I/O prediction when N other threads are performing I/O, then the buffer that corresponds to N is used. In this respect our modelling approach of the I/O subsystem resembles a *flow-equivalent* server (see [17] for example), whose service time depends on the number of queueing clients – here parallel I/O operations.

This methodology generates valid interleavings for threads performing I/O in virtual time. If threads were supposed to issue I/O requests simultaneously in the correct virtual schedule, then this behaviour is replicated by our simulator. Moreover, by keeping profiled values up to date, we may simulate caching effects or temporary network congestion. The disadvantage of this method is that it relies on OS scheduler decisions and real time measurements, which affect the reproducibility of the results. Specifically, as we rely on the OS scheduler to resume a thread T after the completion of the I/O operation, the elapsed time between the system call entry and its exit might include additional time from the scheduling of other threads, i.e. any application or background system threads that get executed before thread T stops its timer. However, the fact that the Linux scheduler temporarily boosts the priority of runnable threads previously waiting for I/O [20] contributes towards limiting such effects.

We note here that both issues we have encountered in this section (i.e. lack of knowledge on whether the I/O system call is served from the cache or not and increased noise in measurements of I/O durations) are a result of our simulation framework functioning at a higher-level than the kernel. There is nothing in principle that forbids us from modifying the Linux kernel code to monitor the lower-level I/O functions, like the `generic_file_read` and `generic_file_write` that only take place when an I/O operation is issued. This is likely to increase the accuracy of the I/O measurements, reduce the VEX profile prediction errors and improve the reproducibility of the simulation results, at the cost of portability, development effort and maintainability.

4.1.3 Misprediction handling

The effectiveness of our methodology to generate a virtual time schedule for "Standard I/O" system calls, that is consistent with the virtual time specifications set depends on the accuracy of the approximation of OVT with PVT. When $PVT = OVT$, [1], then the decisions that the VEX scheduler made at the I/O invocation point, that concerned which other threads should progress in parallel with the I/O system call, are valid; their validity is in comparison to a hypothetical real-time execution. This means that:

- we correctly distinguished a cached I/O from a standard I/O
- in a standard I/O no extra threads were allowed to run in parallel when they should not and no threads were forbidden to run in parallel when they should

Alas, underpredictions $PVT < OVT$ and overpredictions $PVT > OVT$ are a common feature of virtual time I/O. We elaborate on the consequences of each of them for the case where the physical I/O operation has completed and for the one where it has not.

Completed I/O

In the case of underprediction ($PVT < OVT$) of a normal or time-scaled I/O operation, the issuing thread's state is reinserted into the queue of runnable threads Q. This in turn will lead the thread to be rescheduled when its virtual timestamp is the least amongst all runnable threads.

In the case of overprediction ($PVT > OVT$), some threads may already have been scheduled after OVT (but before PVT) which, strictly, violates the virtual time scheduling policy (Figure 4.2). This is the price that is paid for allowing overlapping I/O, but we expect the effect to be small, as the minimised scheduler timeslice only allows short progress for each thread resumed in parallel with the I/O operation.

[1] In practice, since the virtual time scheduler decides on the progress of the simulation threads in a timeslice-based fashion (even a reduced timeslice s_{min} when I/O requests are issued), it suffices that $|PVT - OVT| < s_{min}$

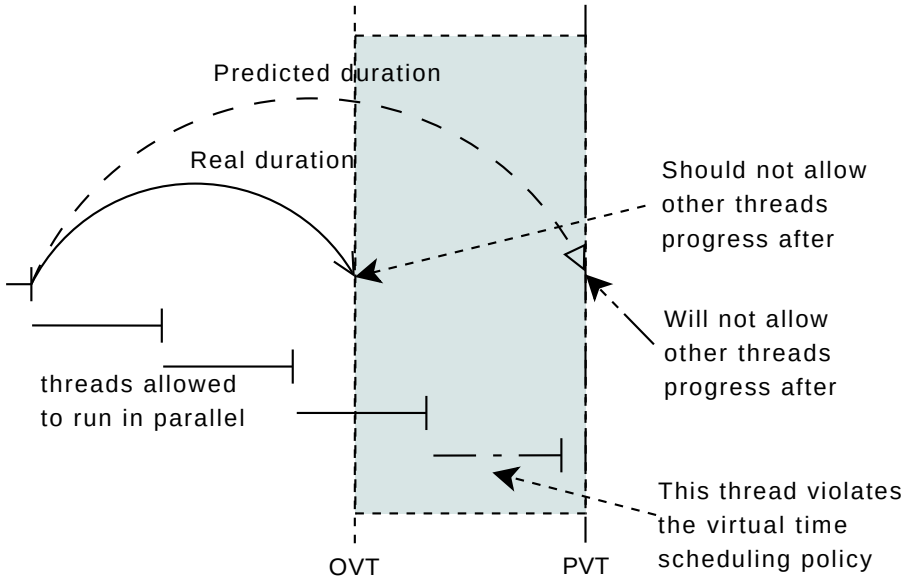


Figure 4.2: I/O overprediction showing the frame where the virtual scheduling policy might be violated

Incomplete I/O

Suppose now that the predicted virtual I/O time (PVT) is reached *before* the I/O has completed, so that the OVT is unknown. In order to determine whether the PVT is an underprediction or not, we keep track of how much *real* time has passed since the I/O request was issued and let $T_{elapsed}$ be the value of the elapsed real time, after any I/O scaling has been applied. $T_{elapsed}$ is thus a *virtual* time. If T_{pred} is the predicted I/O time (duration) then there are two cases that are handled differently in virtual time as follows:

- If $T_{elapsed} \leq T_{pred}$, then an additional (preferably short) period of real time is allowed to pass, which is achieved by the scheduler temporarily using the reduced timeslice s_{min} mentioned in Section 4.1.2 to allow other threads to make progress in virtual time and, crucially, the I/O to make progress in real time.

- If $T_{elapsed} > T_{pred}$, we are sure that $PVT < OVT$ and so are certain of an underprediction. In that case, we update the PVT by an amount of virtual time that is equal to the *total* elapsed virtual I/O time to date. This is tantamount to doubling the current predicted

for it to generate a correct virtual schedule. For the sake of presentation simplicity we assume that correct predictions mean that $PVT = OVT$.

I/O time at each such underprediction point.

4.1.4 I/O scaling

I/O scaling allows us to investigate how the program under test would perform if the durations of its I/O operations were accelerated or slowed down. This would also reflect changes in the functional aspect of the simulation as a result of I/O scaling, e.g. reading a value as it would be updated in virtual time execution, regardless of its physical I/O duration. An I/O operation is scaled when its calling thread has a Time-Scaling Factor (TSF) different than one. What this means is that the total observed duration of the I/O system call (including all levels of the I/O subsystem) is scaled in virtual time. All observed durations are modified according to the *TSF*, before being stored into the prediction buffers: future predictions will be time-scaled, allowing threads to execute in parallel, according to the virtual duration of the I/O operation and not its real one. As *TSF*s are the same for each invocation point, but vary for different invocation points of the same I/O operation, time-scaling works correctly only when combined with prediction at the invocation point level (see Section 4.2): aggregating measurements on the I/O-level, or globally, may mix scaled with unscaled observations.

Standard I/O system calls use the *scaled* predicted duration as the PVT boundary, which determines the threads that are allowed to execute in parallel with the issued I/O operation. The scaled operation is still executed as normal in real time, but its effects on the system are reflected according to the virtual time specifications. This means that values should only be updated by I/O operations, only when the requests were supposed to complete in *virtual time*; not when the physical I/O request is completed.

We illustrate this issue in Figure 4.3-a: thread A issues a read request, which is accelerated so that it completes in virtual time before thread C resumes. Therefore, C should read the updated value of the `buffer` variable, regardless of the duration of the physical I/O operation: this is the value that it would read if the I/O operation was executed faster. Due to the limit imposed by the prediction system (PVT at the dotted vertical line), C is only allowed to

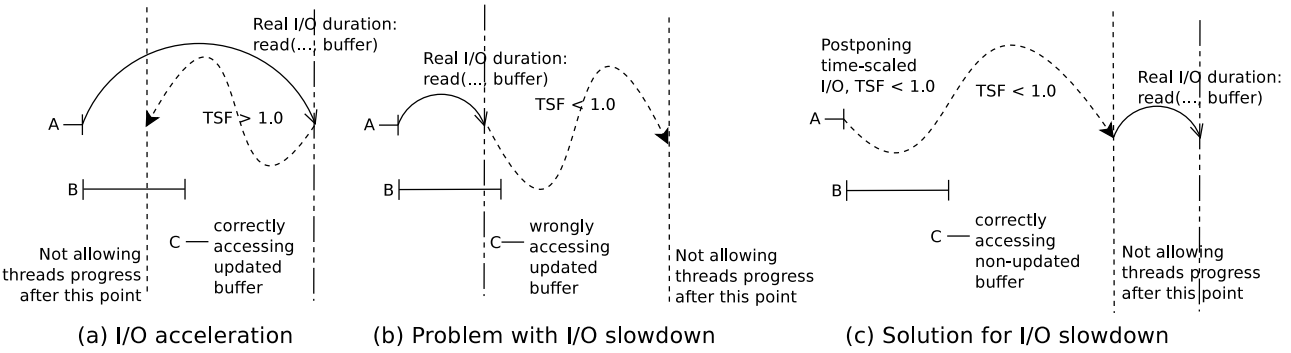


Figure 4.3: Correct I/O scaling in virtual time

continue, after the I/O completes, however long it might take in real time. Therefore, C will read the updated value of the `buffer` as it should do according to the correct virtual schedule.

This does not apply when we are scaling time to increase the duration of an I/O operation. This is depicted in Figure 4.3-b: as the I/O operation completes in real time at OVT before the point PVT where the time-scaled operation is supposed to finish (dotted vertical line), any threads that resume from OVT to PVT, will be accessing the updated `buffer` value. However, if the I/O operations needed TSF^{-1} times longer to execute, then threads should only read the updated value after PVT. This situation is dealt with as shown in Figure 4.3-c: the real operation is issued only when the global virtual time reaches the predicted I/O completion point of the time-scaled operation. This handles the update-value problem correctly, but reduces the potentially parallel contention on the I/O subsystem for decelerated I/O operations and increases the overhead of the simulation. Misprediction for the time-scaled I/O completion point PVT, affect the correct virtual time execution as presented in Section 4.1.3.

It is important to state clearly that time-scaling the duration of an I/O operation is quite a crude approximation, due to the existence of various contention points within the I/O subsystem (buffers, buses etc.) or the use of optimisation techniques for Disk I/O (like merging, scheduling etc. see [62] for details). This means that by scaling the observed I/O time, we scale the delays in each contention point and the performance improvements of any applied optimisations, which is clearly not the same as changing the speed of a single component of the subsystem. The integration of models into the virtual time execution offers a more promising approach towards integrated I/O models [101] or disk simulators [2] instead. However, remaining consistent with

our design target to investigate the effectiveness of a generic high-level system simulator without requesting any specific system models from its users, we left such approaches as future work.

4.1.5 I/O exclusion

An alternative way to investigate the overall performance effect of the I/O subsystem is by selectively excluding the duration of I/O operations and observing the sensitivity of the predictions to this change. Users can provide the method-id of an I/O method to be excluded or a percentage of randomly excluded I/O operations. Once an I/O request is trapped from the higher-level instrumentation interface, the exclusion criteria are parsed to determine whether the system call should be treated as an I/O or not. If its duration is to be excluded, then only the operation's CPU-time is going to be added to the virtual time. The virtual execution is affected accordingly, in that the CPU is not yielded to other threads, whilst an excluded I/O operation is running.

Although the I/O exclusion scheme disregards the elapsed time of an operation, the I/O request is still issued on the system. Accordingly, the behaviour of the I/O subsystem in virtual time remains unaffected by the exclusion of I/O request durations, in that the contention on the resources, caching access patterns, I/O scheduling etc. are not modified. For example, if we issue two I/O requests for the same disk block, the first will lead to an I/O operation and the second will read from the page cache. If we randomly exclude the first request as explained above, then the observed performance of the I/O subsystem will only be defined by the second request, which hits the page cache. With this in mind, the I/O exclusion can still be useful as we show in the evaluation section.

I/O exclusion is compatible with the prediction and threshold schemes used for I/O system calls. If an I/O threshold is defined, the exclusion scheme filters only the operations predicted to be "Standard I/O" operations. Otherwise, it filters all requests. Figure 4.4 shows the handling of the VEX I/O component for the various combinations of I/O thresholds and exclusion parameters. If the prediction P that is generated for each new I/O system call is filtered by the I/O threshold, then the duration of the call is measured in real time, but the CPU is not

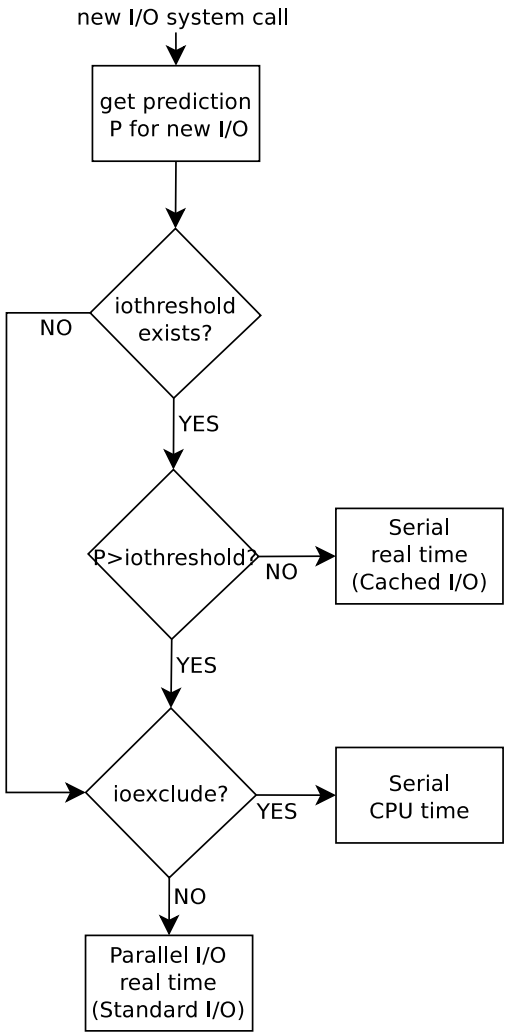


Figure 4.4: Flowchart of I/O in VEX depending on user-defined parameters

yielded to other threads (serial real time); this is the same handling that we use for "Regular" system calls. Excluded I/O operations are also executed sequentially, though their duration is measured in CPU-time (serial CPU-time), so as to ignore whatever happened at the kernel level (including the I/O subsystem). Recognised I/O operations are measured in real time, allowing other threads to execute in parallel (parallel real time), according to our learning and prediction techniques.

4.2 Empirical I/O prediction parameters

VEX implements the methodology for I/O in virtual time, by providing an empirical I/O prediction approach, which stores previous measurements, during a learning phase and then

uses them to generate predictions in the predictive phase. The closer the PVT to the completion time of the upcoming operation OVT, the more similar the virtual execution pattern to the real execution. After the return of the I/O system call and unless otherwise determined by user-defined parameters (see Figure 4.4), the potentially scaled observed time *duration* T_{obs}, is used as the virtual time of the operation. Crucially, the PVT is only used to improve the timing-dependent behaviour of the simulation, whilst the I/O operation is emulated on the host: its value is always replaced in the end by OVT.

In this section we present the various prediction schemes that VEX users can select to improve the approximation of the I/O completion time OVT with PVT and reproduce the real-time behaviour of I/O execution in virtual time. Our approaches are distinguished according to the way that measurements are aggregated and the method that generates new predictions T_{pred} from the previously observed measurements T_{obs}. We selected a variety of prediction methods and measurement aggregation levels; this allow us to perform an exploratory study for the optimal I/O configuration for VEX to evaluate the various configurations and draw conclusions.

4.2.1 Measurement aggregation

The I/O module of the simulation supports three levels of prediction, depending on the way that previous measurements of I/O requests are grouped together. The most abstract is the *global* prediction level, where all I/O calls share the same prediction-supporting data structures. Depending only on the number of threads issuing I/O operations in parallel N, all profiled durations are pushed into a common data structure for N threads, as mentioned in Section 4.1.2. The lower level aggregates measurements by the method-id of the I/O operation (*ioop* level), while the lowest level maintains a different buffer for each I/O invocation point (*invpoint* level), identified by an invocation-point id provided by the upper instrumentation layer. The idea of aggregating I/O measurements at the *global* and global *ioop* levels was also addressed in [50], where partitioning I/O measurement data in global and read/write categories yields accurate prediction results.

The trade-off involved is that the lower the aggregation level, the higher the matching of the prediction to the duration of each operation and the longer the time until we enter the predictive phase. To provide a PVT at the *invpoint*-level for a defined buffer size of B measurements, when N threads are executing I/O in parallel, we would need at least B previous results from that I/O invocation point for exactly N threads. In the case of *global*-level prediction any B measurements are enough. By remaining in the learning phase (gathering measurements to fill a buffer) for a longer time, we reduce the chances of creating thread interleavings consistent to the virtual time execution, but potentially improve the accuracy of the prediction.

The number of parallel threads N is defined upon entry to each I/O operation: we are using the total count of I/O threads at the time that the operation started. This reflects the contention on the I/O subsystem before the request is issued, which in turn affects its real time duration. VEX supports a single global counter of threads performing I/O in parallel. We note that there is no guarantee that the N threads contend to access the same resource; for example, they might be reading files from different disks or different devices. Here, we assume a single I/O resource. The identification of the resource that each I/O operation targets can be facilitated by a kernel-level VEX which is left as future work (see Section 9.3.1).

4.2.2 Prediction methods

All prediction methods used by VEX follow the same general approach, in that they store OVTs (irrespective of whether they are recognised as Cached or Standard I/O operations) until a buffer is full. When the prediction buffer is full, the next measurement is estimated, according to some statistical or modelling method. The OVTs of operations whose durations were originally predicted are still pushed back into the buffer (according to the I/O prediction level). The buffer size B can be parameterised by the user: larger buffers contain samples that cover a wider period of time, but delay the beginning of the predictive phase, for the number of threads N at the selected I/O level.

We review the prediction policies supported by VEX, which treat the underlying I/O subsystem as a black-box. This reflects our design choice to refrain from modelling the structure of the

entire I/O filesystem [101, 121], thus maintaining VEX's portability and independence from the simulation host's system.

Sliding-window replay

When the buffer is full, the first stored result for the duration of an I/O request is used unmodified as a prediction. The T_{obs} of the next I/O operation will replace the predicted value T_{pred} that was used to define PVT. This is tantamount to a cyclic buffer and simulates a sliding-window effect. According to this *replay* prediction scheme, the distribution of the T_{obs}s is exactly reproduced by the generated T_{pred}s. Nevertheless, this approach is subject to high overprediction errors, when the T_{pred} corresponds to an I/O operation and the next operation hits the page cache. High underprediction errors are expected to occur in the opposite case.

Basic statistical modelling

To investigate different trade-offs in the handling of extreme T_{obs}s, we propose a series of simple statistical methods on the previous B measurements. The *min* method always uses the minimum value of the last B measurements. This favours predictions to be perceived as cache accesses, leading to an I/O request only if each of the last B measurements correspond to one. Conversely, the *max* method results in more parallel I/O operations, because a single I/O request in the last B measurements is enough to predict a long I/O duration for the new operation. This, in turn, allows more threads to execute in parallel, leading potentially to an increased load on the I/O subsystem. Consequently, *min* increases the number of underpredictions and *max* the number of overpredictions. Using the average of the last B measurements (*avg* method), creates a balance between the two extremes. To filter out the effect of extreme values altogether, VEX can be configured to return the *median* value of the last B T_{obs} samples.

Distribution modelling

Under this category we group all approaches that use previous samples to generate a distribution of the T_{obs}s of the I/O operations. In the *sampling* prediction scheme, the last B measurements are assumed to belong to a positive truncated normal distribution and a random value based on the mean and standard deviation of this sample is generated. To model the fact that I/O measurements may create clusters, depending on whether the request hit the kernel buffer, the on-disk buffer or just missed, VEX can be set to describe the duration of each request as an M-state Markov chain. Oly et al [102] use Markov chains to describe the spatial patterns of I/O requests in scientific workloads. Here, a similar idea is explored for probabilistically modelling patterns of I/O durations. The transition probabilities of the Markov chain are computed by partitioning the last B measurements in exactly M ranges. The transition probabilities are then calculated by observing the sequence of the last B measurements. We note here that there is no guarantee that the resulting Markov chain will be irreducible: for example if the first value of the last B T_{obs}s defines its own range that is never returned to within the next $B-1$ measurements, then the resulting chain will not be irreducible. Once a new prediction T_{pred} is requested to determine PVT, we return a randomly selected previous sample from the current state and move to the next state, according to the generated transition matrix. Accordingly, the *M-markov* method can be perceived as a selection method between M different random replay predictors. To model possible temporal effects of the I/O subsystem, a new Markov chain is generated every B predictions.

An alternative configuration of VEX is to utilise a GSL [48] histogram, generated by a previous (instrumented) run. To exclude simulation effects on the I/O subsystem, this "I/O-profiling" execution can disable the scheduler of VEX, thus yielding the control of threads to the OS. The measurements are exported according to the I/O-level (to a single file for all operations, to one file per I/O type or to one file per invocation point) and then imported by the VEX simulation run. The probability distribution of each GSL histogram determines the next T_{pred}. Though the "*gsl*" approach captures the entire I/O behaviour for the selected workload, it does not model temporal effects or behaviours not present in the trial run (e.g. extra load on the I/O

system).

Time-series modelling

In our investigation of empirical I/O modelling during runtime, we also include a few simple prediction techniques from time-series analysis, which have been used for similar purposes in [151]. The *cmean* method, returns the mean of all measurements up to this point for the selected I/O level, being updated after every I/O request. The *linregr* performs linear regression on the last B samples, with time being the independent parameter: measurement i in the buffer happened at time t_i for $i = 0, \ldots, (B-1)$. Inserting a new measurement M results in shifting value i to $i-1$ for $i = 1, \ldots, (B-1)$ and storing M in the position (B-1) of the buffer. After calculating the slope and intercept for the sample and generating the function $y(t)$, we return value $y(B)$ as the next prediction.

VEX also supports autoregression (AR) analysis techniques, that use the last B measurements in the buffer b_0 to b_{B-1} to generate the autoregression coefficients α_i. The prediction PVT is then generated by: $PVT = \sum_{i=1}^{B} \alpha_i * b_{B-i} + \epsilon$. To calculate the coefficients, we integrated an open source version of the Burg Maximum Entropy algorithm, found in [19].

For future reference all prediction methods are tabulated in Table 4.1.

4.3 Evaluation

The purpose of this section is to evaluate the methodology for including I/O in a virtual time simulation. We first present our two custom I/O-stressing benchmarks, which were set-up in such a way as to enable us to highlight differences in the measurement aggregation schemes. Using these benchmarks we show the necessity of an I/O module. Then we probe various configurations for the I/O prediction in an attempt to identify an optimal configuration, that consistently yields the most accurate results. We use the findings of this study to evaluate I/O effects provided by VEX, like I/O scaling and I/O exclusion. We complete our evaluation

Category	Method	Description
Sliding-window replay	replay	Reuse sample taken B measurements ago
Basic statistical modelling	avg	Use average of last B measurements
	max	Use the maximum of the last B measurements
	median	Use the median of the last B measurements
	min	Use the minimum of the last B measurements
Distribution modelling	sampling	Sample from the normal distribution generated by the last B measurements
	M-markov	Reuse sample from the current state and randomly transition to the next state according to a Markov chain of M-states (default 2) generated every B measurements
	gsl	Reuse sample from previous execution (with VEX scheduler disabled) described by a GSL histogram
Time-series modelling	cmean	Use average of all measurements up to this point
	linregr	Use $y(t_B)$ where y(t) is a linear regression of the last B measurements treated as the dependent values from $t_0 ... t_{B-1}$
	arburg	Use $y(t_B)$ where y(t) is an autoregressive (AR) function solved by the Burg method

Table 4.1: Virtual Time I/O prediction methods

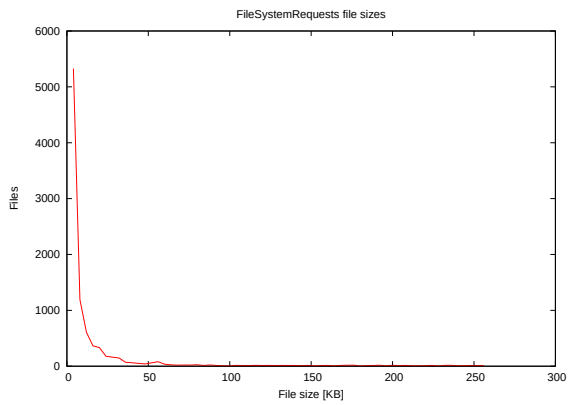
with an experiment on a database-oriented benchmark, using two different database servers and connectors. We conclude with the findings of our study.

4.3.1 Benchmarks

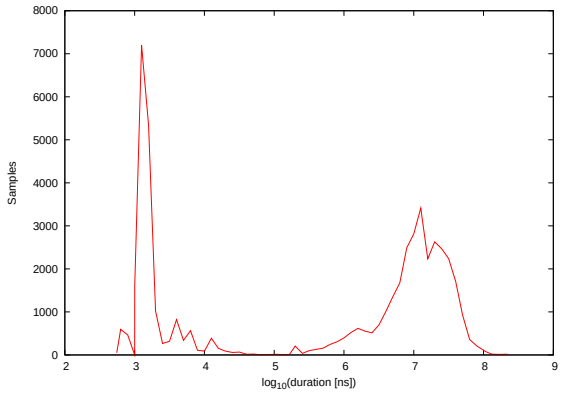
We introduce two benchmarks: the "Disk I/O" and the "No-Memory I/O", which are small programs that stress the disk I/O subsystem. Both are written in Java and communicate with VEX via the upper layer called the Java Instrumentation Environment (JINE), which is presented in Chapter 5. At this point it suffices to say that the role of JINE is to trap the I/O system calls at the level of the Java system classes that invoke them and assign ids to each system call and each invocation point (for details see Section 5.1.2).

Disk I/O benchmark

In this stress benchmark we have N threads that read R files from a list of F files in total and print into a common file the checksum of their contents by parsing them in blocks of B bytes. Each file in the list is randomly selected amongst all files that are stored in any subdirectory



(a) File sizes of the real files used in the I/O-stressing benchmarks. The average is 18.87KB and the median and mode 4KB



(b) Histogram of all observed I/O measurements

Figure 4.5: Measurements on the workload of the Disk I/O benchmark

of arbitrary depth under the root '/' directory of the simulation host. In contrast to the case where all files are stored in the same path, this approach also takes into account the performance impact of the directory cache. In this benchmark, called "Disk I/O", the list of files is loaded entirely into memory. To avoid effects from previous runs, the I/O system caches are flushed and cleared before each experiment in real and virtual time. The sizes of the random files found on the disk of the simulation host range from 4 to 256KB according to figure 4.5a. A histogram of the I/O durations acquired via VEX with the scheduler disabled in a "pure profiling" run (see Section 3.2.4) is illustrated in Figure 4.5b, showing how measurements are clustered in a "Low" and a "High" cluster depending on whether the page cache is hit or not. We assume that only measurements from the "High" cluster would trigger DMA in a real-time execution.

Parameter	**Description**	**Value**
N	Threads	16
R_{mem}	Files to be read when list is in memory (Disk I/O)	1000
R_{disk}	Files to be read when list is on disk (No-Memory I/O)	10
F	Total files in list	9287
b	Bytes to be read each time from a file	128K
I_T	I/O threshold	50μs
B	Size of each prediction buffer	16

Table 4.2: Parameters used for Disk I/O benchmark

Our tests for this benchmark have been performed on Host-1 (see Appendix A). Table 4.2 shows the configuration parameters selected for this benchmark. N was selected to allow several threads to perform I/O in parallel, while the R_{mem} and R_{disk} parameters were chosen

to keep the running time limited. The total number of files F was the one found under the '/' directory of the simulation host, after selecting an arbitrary large number of files (here 10,000) and removing all symbolic links. The bytes per I/O b are meant to trigger longer I/O durations and read most files in one system call. The I/O threshold I_T is set according to the clusters of observed I/O durations found in the histogram of Figure 4.5b, to distinguish between Standard and Cached I/O operations. B is set to 16.

No-Memory I/O benchmark

To investigate the effect of aggregating measurements at different granularities (see Section 4.2.1), we change the Disk I/O benchmark, so that the list of files is not loaded into memory, but accessed directly from the disk in each of the R requests. Although this does not make sense from a performance perspective, it helps with the investigation of the I/O prediction scheme, by increasing the load on the disk and creating a set of I/O requests that have a higher probability of being found in the page cache (the ones accessing the list).

No I/O in virtual time

Our validation approach concerns executing the "Disk I/O" benchmark in both real and virtual time (without any time-scaling) and comparing the results for various prediction methods. The last row of Table 4.3 ("No I/O") shows the VEX prediction when the durations of all I/O operations are excluded. This approach is enforced with the "iodisregard" parameter of the VEX simulator (i.e. I/O-exclusion of 100% of the I/O operations) , which only uses CPU-time measurements and serialises all system calls. As we disregard all I/O durations in an I/O-bound stress test, we acquire a very high underprediction (-94%) for the total response time of the program predictions. This clearly illustrates the need for a special module to handle I/O in virtual time.

4.3.2 Optimal configuration investigation

In this section we investigate how the various configuration choices of the I/O module of VEX affect the total virtual response times of our benchmarks and determine whether there is an optimal configuration for the prediction scheme of the simulator. Specifically, we execute our benchmarks with each of the supported prediction policies (see Section 4.2.2), while changing the following parameters:

- I/O threshold: deciding whether to use an I/O threshold to distinguish between Standard and Cached I/O operations or not.
- Prediction aggregation: changing the level at which measurements are aggregated in the same prediction buffer.
- Size M of the Markov-chain: varying the number of states of the Markov-chains that are used to model the buffered I/O measurements in the *M-markov* prediction method

I/O threshold usage

The rows of Table 4.3 present the VEX predictions when all I/O system calls are treated as Standard I/O. We observe that the predicted benchmark total time is slightly (≈4%) to extremely (≈75%) lower than the corresponding real one. What happens in this case is that every time an I/O system call occurs, it is recognised as "Standard I/O" and thus notifies the VEX scheduler to allow another thread to progress in parallel. Accordingly, all system calls allow for parallel execution of code that would otherwise be serially executed in real time; this results in low predicted times.

The results of Table 4.3 show an inverse correlation between the average number of threads issuing I/O in parallel (fourth column) and the (absolute) prediction error: the higher the parallelism the closer the prediction to the observed benchmark time. Similarly, the higher the percentage of underpredictions the higher the absolute prediction error. Underpredictions lead to limited parallelism between threads that are actually performing disk I/O: the VEX

Method	Error (%)	Overhead	Parallel I/O	Under (%)	Correct (%)	Over (%)
max	-4.53	1.09	14.90	19.11	9.41	71.48
cmean	-11.20	1.15	13.88	23.55	10.28	66.16
arburg	-18.00	1.14	11.51	37.03	7.67	55.30
avg	-18.53	1.15	11.29	27.76	9.19	63.05
sampling	-27.33	1.16	9.84	28.88	9.21	61.91
linregr	-47.80	1.10	7.29	35.63	6.94	57.44
median	-62.13	1.16	5.03	39.57	19.28	41.14
replay	-63.64	1.14	4.81	41.15	11.38	47.47
gsl	-65.40	1.19	4.58	40.92	10.77	48.31
markov	-67.80	1.14	4.39	38.55	21.33	40.12
min	-74.60	1.13	3.69	63.88	35.91	0.21
No I/O	-93.79	1.29				

Table 4.3: Results for the Disk I/O benchmark without an I/O threshold. "Correct" predictions are defined to be within a 20% (absolute) error of the observed I/O real time. Means of 15 measurements shown.

scheduler does not resume runnable threads that are suspended at a virtual timestamp higher than the under-predicted I/O completion and so limits the chances of parallel I/O occurring. In turn, this limits the contention on the I/O subsystem, resulting in under-predicted I/O operations yielding lower observed I/O durations, than the ones that are found in the real time execution of the benchmark. This leads to a vicious cycle of underprediction of I/O durations and ultimately to the lower predictions for the total benchmark time on the lower half of Table 4.3. The way that underpredictions affect the subsequent observed real time durations T_{obs} is shown in Figure 4.6: the observed I/O times of the VEX simulation with the *min* prediction method, which exhibits the highest percentage of underpredictions, provide significantly lower T_{obs}s than the *max* or the real time ones. In fact the latter two almost overlap, thus explaining the relatively high accuracy of the *max* prediction method. Accordingly, the prediction methods that provide the best results are the ones that mostly factor in the few long lasting I/O operations (*max*, *cmean*, *avg*), generating higher I/O predictions for the next I/O duration. These in turn allow for many threads to issue I/O operations in parallel, leading to increased contention on the subsystem and thus yielding again high T_{obs}s, perpetuating this effect. An unexpected result is that it seems more important to avoid underpredictions than predict correctly (within a 20% absolute error). Interestingly, the *min* prediction policy with the highest correct prediction percentage provided practically unusable results, despite the fact that it had the highest percentage of "correct" predictions.

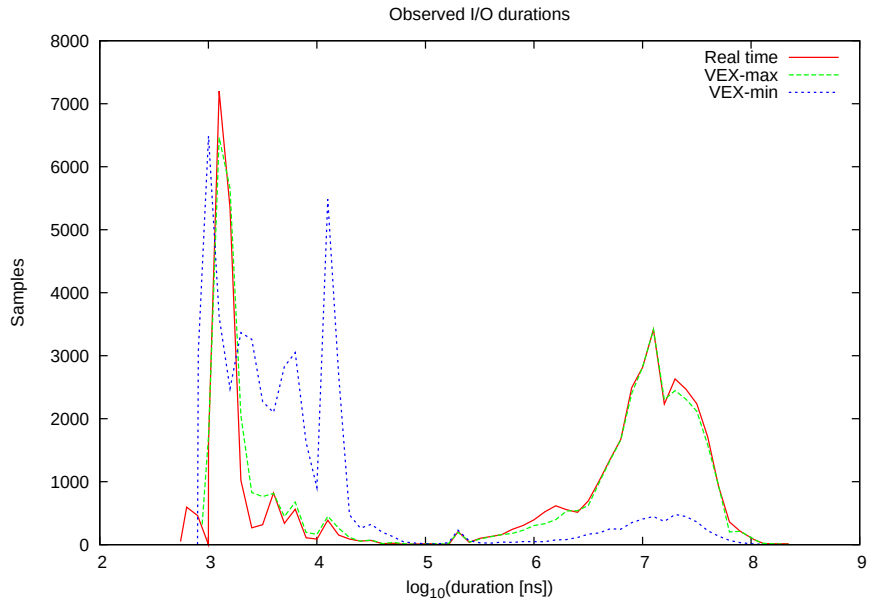


Figure 4.6: Histograms of the *observed* I/O durations in the real execution and in the virtual time executions with the *max* and *min* prediction policies

We next enable the I/O threshold parameter I_T as the limit over which predictions trigger the scheduler to treat the next I/O system call as a Standard I/O request. If the next prediction for a system call is lower than I_T, then it is serially executed and its duration is measured in real time. The results for this configuration are shown in Table 4.4 and present an improvement from an average error of -41.91% between all prediction methods in the previous setting to 9.88% in this one. In contrast to the experiments without an I/O threshold, all predictions here are higher than the ones of the real time experiments. We also observe again a positive correlation between the underprediction percentage and the prediction error.

The effect of underprediction when the I/O threshold is enabled, differs in that no I/O parallelism whatsoever is allowed. The system call is (wrongly) predicted to hit the cache and so is does not yield the processor to other threads. The difference with the previous configuration is that, in that case, all operations were still performed in parallel, but not to the extent of the real-time execution – here they just run one after another. This increases both the predicted virtual time as well as the total execution time of the simulation, as seen in the results of Table 4.4, explaining the strictly positive prediction errors observed. This observation is also partially attributed to the noise in real time measurements on the simulation host, due to background system load.

Method	Error (%)	Overhead	Parallel I/O	Under (%)	Correct (%)	Over (%)
max	5.96	1.34	13.21	13.99	8.18	77.83
gsl	8.38	1.41	4.35	33.55	31.82	34.64
cmean	8.50	1.25	10.38	20.80	10.19	69.02
replay	9.13	1.45	4.28	33.75	31.47	34.78
markov	9.75	1.44	4.23	33.02	32.27	34.71
linregr	9.92	1.38	5.86	31.73	17.53	50.74
arburg	10.46	1.35	11.04	36.71	10.33	52.96
sampling	10.71	1.30	7.17	29.07	14.72	56.21
avg	10.88	1.38	6.68	27.22	14.88	57.90
min	11.54	1.36	6.60	59.62	39.82	0.56
median	13.50	1.33	6.75	42.49	13.12	44.38

Table 4.4: Results for the Disk I/O benchmark using an I/O threshold of 50μs. Means of 15 measurements are shown.

Table 4.5 summarises the observations and conclusions for the effects of over- and underpredictions based on the presented results.

Actual	Predicted	No I/O recognition	I/O recognition (threshold)
Low	Low	Incorrectly runs in parallel with others: lower total time	Correctly running alone and measuring only its real time: no effect (apart from external noise)
Low	High	Incorrectly runs in parallel with others – overprediction fixed upon exit from I/O: potentially lower total time (depending upon overprediction identification overhead)	Incorrectly runs in parallel with others – overprediction fixed upon exit from I/O: potentially lower total time (depending upon overprediction identification overhead)
High	Low	Correctly runs in parallel with others – underprediction limits parallelism: lower time	Incorrectly does not run in parallel with others – serializing long I/Os and increasing total time
High	High	Correctly runs in parallel with others: no effect (apart from external noise)	Correctly runs in parallel with others: no effect (apart from external noise)

Table 4.5: Issues in virtual time I/O with and without an I/O threshold. "Low" times are supposed to hit the cache and run sequentially, while "High" ones are expected to perform an I/O with DMA and allow other threads to make progress in parallel

Prediction aggregation

Up to this point all I/O predictions were aggregated globally (see Section 4.2.1). Recall that measurements can either be aggregated globally for all I/O operations, per I/O operation or per different invocation point. This is due to the setup of the previous experiment where the vast

majority of I/O operations concern the same operation (`FileInputStream.readBytes`) and the same invocation point (where the file is parsed to create its checksum). Thus no differences exist among the various levels of aggregation. In this section we use the "No-Memory" benchmark, that has two points with increased I/O activity: one where a file name is retrieved from disk and one where the file is parsed block by block.

Table 4.6 presents the results for the various prediction methods and aggregation levels, when the I/O threshold scheme is set. The table shows a clear improvement from the transition from the single global buffer to the buffer per I/O or to the per invocation-point scheme. Indeed, the latter two cases offer a better distinction between the I/O operations reading the file list (possibly cached and short) and the ones reading the contents of the files (possibly uncached and long). On the invocation point prediction level the *markov* method presents the best results with a mean error of 6.5%. We note here that the overhead is significantly higher than the case where the list of files is loaded into memory (Disk I/O benchmark), due to the increased number of I/O system call invocations used to parse the list of files for each request (from approximately 50,000 to 900,000).

	Single global buffer		**Buffer / I/O operation**		**Buffer / Invocation Point**	
Method	**Error (%)**	**Overhead**	**Error (%)**	**Overhead**	**Error (%)**	**Overhead**
arburg	7.83	5.70	5.00	5.50	7.33	5.85
avg	49.50	4.46	11.67	7.17	10.25	6.55
cmean	19.50	3.79	15.25	7.12	7.67	6.10
gsl	14.17	6.98	10.75	6.96	8.00	6.00
linregr	50.67	5.52	11.25	6.76	6.67	6.92
markov	9.88	5.26	9.64	5.41	6.51	5.23
max	7.50	5.55	5.83	6.07	13.25	6.76
median	88.33	2.86	26.75	6.64	17.25	7.64
min	83.50	3.19	24.50	6.28	39.25	5.72
replay	64.50	6.07	10.67	7.24	8.75	6.31
sampling	41.21	9.39	12.64	6.17	8.36	7.87
Mean	*39.69*	*5.34*	*13.09*	*6.48*	*12.12*	*6.45*

Table 4.6: Effect of prediction aggregation on the "No Memory" I/O benchmark, when the list is always read directly from disk

Based on the presented results, it is not possible to identify a prediction method that consistently outperforms all others. For example, *max* which yields good predictions for this benchmark, presents a slightly increased error when measurements are aggregated per invoca-

tion point. The only clear conclusion is that prediction methods like *min* and *median*, that favour underpredictions, by filtering out higher values typically provide bad predictions for such I/O stress-tests. The influence of the aggregation level of predictions depends on the variety of I/O operations and behaviours per invocation point in the VEX simulated program.

Markov chain size

For a constant buffer size of $B = 16$, we investigate the effects of different parameters M for the $M - markov$ prediction method, i.e. of different number of states (in this context levels) that describe the observed I/O values for each buffer. To visualise the effect of M, we export the observed I/O durations of the "Single global buffer" for the disk I/O benchmark that loads the file list into memory and experiment with an offline tool (C++ program) that applies the algorithm that generates the Markov chain to these values. Figures 4.7a - 4.7e illustrate the Markov chains generated for a particular subset of $B = 16$ measurements. All figures have a state at the 15.9ms level, which is never returned to, which corresponds to the first measurement of the set. Its existence leads to the creation of a quite wide state in the $2 - markov$ case (Figure 4.7a, ranging from 1.1μs to 2.9ms. As we increase the states this state is divided into a number of sub-states, leading to all states for $M > 4$ to have a maximum range of 0.5ms with an extreme of 0.1ms for $M = 8$ (Figure 4.7e).

However, the effect of the chain size to the prediction accuracy is small (within 4%) for both experimental configurations presented, as shown in Tables 4.7 and 4.8, for the memory loaded and disk loaded file list respectively. Only in particular cases, there might exist a statistically

	Single global		**Per I/O operation**		**Per invocation point**		**Total**	
M	**Error(%)**	**Stdev**	**Error(%)**	**Stdev**	**Error(%)**	**Stdev**	**Error(%)**	**Stdev**
2	3.89	4.44	7.22	5.15	16.33	5.27	9.15	8.78
3	7.44	6.78	13.44	4.66	3.22	4.98	8.04	8.72
4	5.11	5.01	16.00	4.88	12.67	5.15	11.26	10.29
6	3.11	1.45	5.56	5.15	9.75	5.12	6.00	7.08
8	3.67	1.73	9.89	4.93	10.50	5.35	7.92	7.97
Mean	4.64		10.42		10.49			

Table 4.7: Effect of chain size in the *markov* prediction policy for the Disk I/O benchmark with memory-loaded file list. Means and standard deviations of 9 measurements on Host-1

	Single global		Per I/O operation		Per invocation point		Total	
M	**Error(%)**	**Stdev**	**Error(%)**	**Stdev**	**Error(%)**	**Stdev**	**Error(%)**	**Stdev**
2	9.45	9.65	9.00	10.91	4.55	8.49	7.63	10.00
3	12.75	11.82	9.63	11.81	8.63	10.45	10.33	11.93
4	11.25	9.91	9.25	9.90	7.25	8.96	9.25	10.04
6	8.00	10.50	11.75	13.32	5.00	7.86	8.25	10.63
8	8.13	7.30	8.75	9.96	7.88	10.16	8.25	8.93
Mean	9.92		9.68		6.66			

Table 4.8: Effect of chain size in the *markov* prediction policy for the Disk I/O benchmark with disk loaded file list. Means and standard deviations of 9 measurements on Host-1

significant difference between two values for M: performing the t-test for the "Single global buffer" aggregation level of the "memory" test, we find that the difference of the prediction error between $M = 3$ and $M = 6$ is statistically significant with 95% confidence. In this case, globally clustering the values in 6 levels outperforms doing so in 3, which we assume is attributed to the characteristics of the program. Identification of such charatecteristics at runtime to improve prediction accuracy (as in [151]) is left as future work.

4.3.3 Investigating I/O effects

Our focus up to this point has been to investigate the effect of the various prediction methods and aggregation levels of VEX to the observed application real times. In a sense, we could regard this as the configuration stage of our I/O modelling approach: for the No-Memory Disk I/O benchmark the *max* prediction method with one buffer per I/O operation has provided the lowest prediction errors, due to the high number of I/O duration overpredictions. We can now use this configuration of the VEX simulator to identify the effect time-scaling and I/O exclusion. We note here that such an extended "tuning" phase is not necessary; users can typically use *max* that is bound to favour overprediction. However, as our previous study has shown, this type of tuning can be beneficial to the accuracy of the virtual time execution.

Time-scaling

We identify the effect of time-scaling in the No-Memory Disk-I/O benchmark, by modifying the time-scaling factor (TSF) of a method that invokes I/O operations: in this case we choose the method that parses files to get their checksums. The x-axis of Figure 4.8a shows the ratio $\frac{1}{TSF}$, increasing it by a step of 0.25. The acquired results of Figure 4.8a show a non-linear relation between the method $\frac{1}{TSF}$ and the VEX predicted program execution times. We clarify here that it is not possible to verify these results without using some kind of file system simulator - even then it would not be clear how to configure the simulator to get the equivalent effect of VEX time-scaling.

An interesting observation regards the overhead of the VEX simulation shown in Figure 4.8b, which remains constant for $TSF > 1.0$, but becomes up to approximately 13 times higher for $TSF < 1.0$. This is explained by the delayed I/O execution scheme that occurs when we are decelerating time ($TSF < 1.0$), under which an I/O execution is delayed until after the (time-scaled) predicted I/O virtual timestamp is reached (see Section 4.1.4 for details).

I/O exclusion

In this experiment we use the same configuration to experiment with the effect of I/O exclusion, which allows us to investigate the effect of reducing the number of I/O operations. A reduction in the prediction results is expected in any case, because we replace real time with CPU-time measurements for all excluded operations. The I/O exclusion simulation thus provides an insight to the dependence of the overall performance from I/O. Indeed for our I/O-bound benchmark, we observe in Figure 4.9 that the effect of I/O removal is significant and also not linear: removing 60% random I/O operations reduces the total predicted time by more than 80%.

An interesting observation on Figure 4.9 is that the variance of the predictions is quite high for the exclusion of a low random percentage of I/O operations (10% and 20%). Based on our observations for the existence of two large clusters of I/O durations, if we exclude only a

few samples, it is likely they all belong entirely to one or the other cluster. As we increase the randomly excluded I/O operations, the probability that we exclude an adequate number of samples from both clusters increases and the variability of the results is reduced.

4.3.4 Database prediction

We now present a setup to evaluate how the combination of VEX and JINE handles database requests. We have created a sample database, using the original database-populating class from the TPC/W-benchmark [49]. We spawn 16 client threads that each connect to the database through the Derby JDBC driver. Once connected, every thread thinks for a random amount of time Z (sampled from an exponential distribution) and then sends N randomly selected SQL queries (from a set of pre-defined query templates) to the server. We vary the thinking time Z and measure the average response times in real time. We then execute the client with VEX/JINE with adaptive profiling turned on and explore two variants of the code: an unmodified version with no scaling and a version of Derby's heavily used method (`org/apache/derby/jdbc/Driver40.newEmbedPreparedStatement`) (the *target* method), scaled by a factor of two. This is expected to reduce contention on the compilation and optimisation schemes of the Derby SQL engine. To avoid warmup effects the procedure is repeated within the same instance of the JVM until the results converge.

Derby DB

We use the Derby database server [4] which is fully implemented in Java and so that all operations within the server can be monitored and controlled by the VEX scheduler, after being instrumented by JINE. The results are shown in Figure 4.10 for N=60. This shows the measured (real) time and the predicted virtual times with and without scaling. Also shown ("Profiling") is an additional estimate of the scaled performance, which is computed using the proportion of total time spent in the target method, as measured during the unscaled execution within the framework. This is precisely the information that would be provided by a

conventional profiler. Although there exists an outlier for a think time of 64ms, the predicted and observed times agree reasonably well, with an average prediction error of 16.5%. Scaling of the target method evidently has a non-linear effect on the overall performance that cannot be accounted for straightforwardly by conventional profiling. Unfortunately, verification of the accuracy of the scaled performance prediction in this context is not feasible without elaborate software performance modelling or simulation techniques, a fact that underlines the purpose and usefulness of our framework.

MySQL DB

We repeat the experiment of the previous Section 4.3.4, but replace the the database server with MySQL, connecting to it via the MySQL-J connector driver. Table 4.9 shows the average prediction errors for the server response time for seven prediction policies at the I/O invocation point level. We observe that, once again, the results that have used an I/O threshold (here 50μs), perform significantly better than the ones that haven't. Policies *max* and *arburg* remain amongst the most accurate. As shown in Figure 4.11, *arburg* converges to the observed real time as the average waiting time increases and the load on the server stabilises.

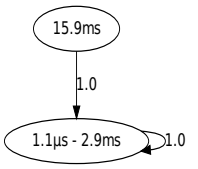
Prediction method	**Error** [%]	**Error with I/O threshold** [%]
arburg	12.38	5.97
avg	29.28	11.75
linregr	25.97	7.78
markov	31.70	10.32
max	23.40	7.95
replay	30.34	3.23
sampling	38.33	13.06

Table 4.9: Average prediction errors for the MySQL database experiment with and without the use of an I/O threshold as a function of the average think time Z

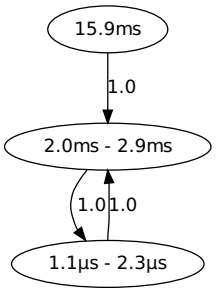
Overall we can summarise the findings of our evaluation study of the I/O module of VEX as follows:

- To achieve a meaningful prediction accuracy for I/O-bound benchmarks it is absolutely essential that I/O durations be included in virtual time.

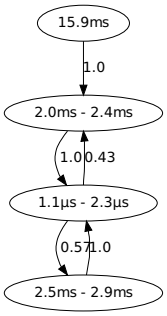
- Definition of even an approximate I/O threshold to distinguish between Standard and Cached I/O operations improves the prediction results of the framework.
- I/O simulation with a threshold yields higher predictions for the total application time, which can be attributed to the serialisation of I/O operations on I/O-underpredictions and the noise incurred by real time measurements.
- No prediction method and I/O module configuration has been found to be consistently better than any other.
- Prediction methods that favour overprediction of the duration of the next I/O operation (e.g. *max*) perform better than the ones who favour underprediction by allowing a higher level of parallel I/O.
- In cases where various invocations of the same I/O operation constantly produce different I/O durations (e.g. the No-Memory I/O benchmark), then specifying a lower level at which measurements are aggregated is beneficial to the prediction of the framework.
- Effects like the length of the chain in the *markov* prediction method have not been found to have a significant effect on prediction accuracy.
- I/O scaling and I/O exclusion can disclose non-trivial effects on the dependence of the total response time to I/O.



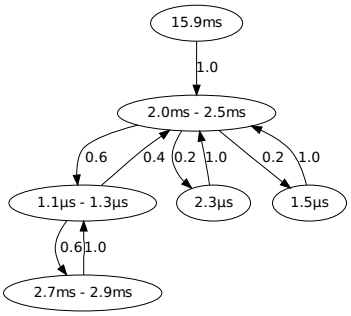
(a) $2 - markov$



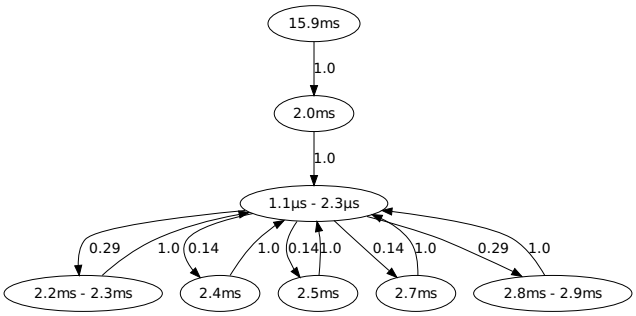
(b) $3 - markov$



(c) $4 - markov$

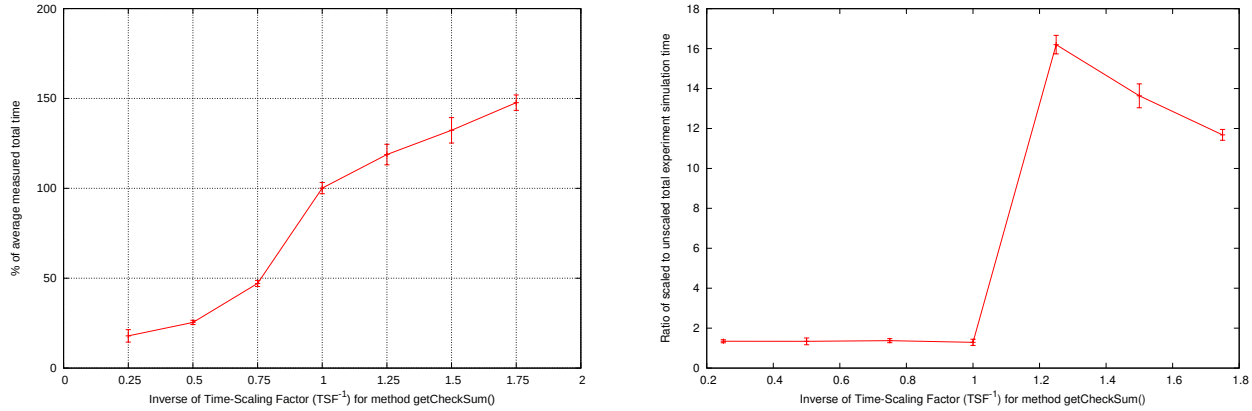


(d) $6 - markov$



(e) $8 - markov$

Figure 4.7: Visualisation of the effect of M in modelling I/O durations in a particular buffer set of $B = 16$ measurements with the $M - markov$ prediction method



(a) Prediction (b) Overhead

Figure 4.8: Virtual time prediction as a rate to the (unscaled) real execution and overhead of the Disk I/O "memory" benchmark for various time-scaling factors (TSF). Means and (approximately) 95% confidence intervals of 7 measurements are shown

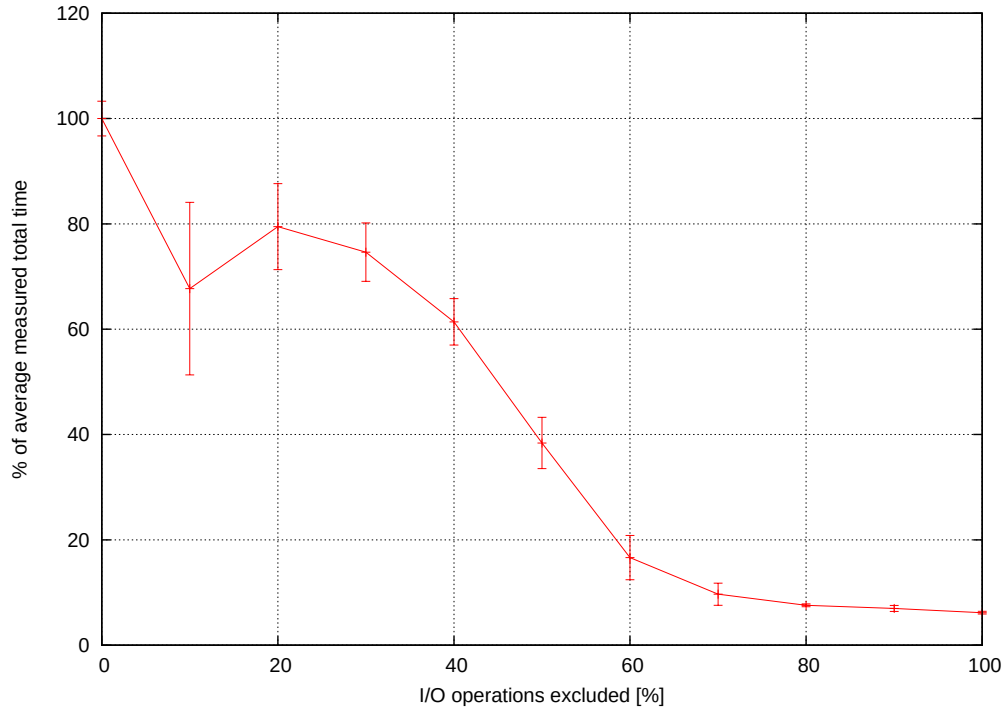


Figure 4.9: Effect of I/O-exclusion in the Disk I/O "memory" benchmark (with a threshold of 50μs). The x-axis is the percentage of recognised I/O operations, whose duration is set to be replaced by CPU-time. Means and (approximately) 95% confidence intervals of 10 measurements are shown

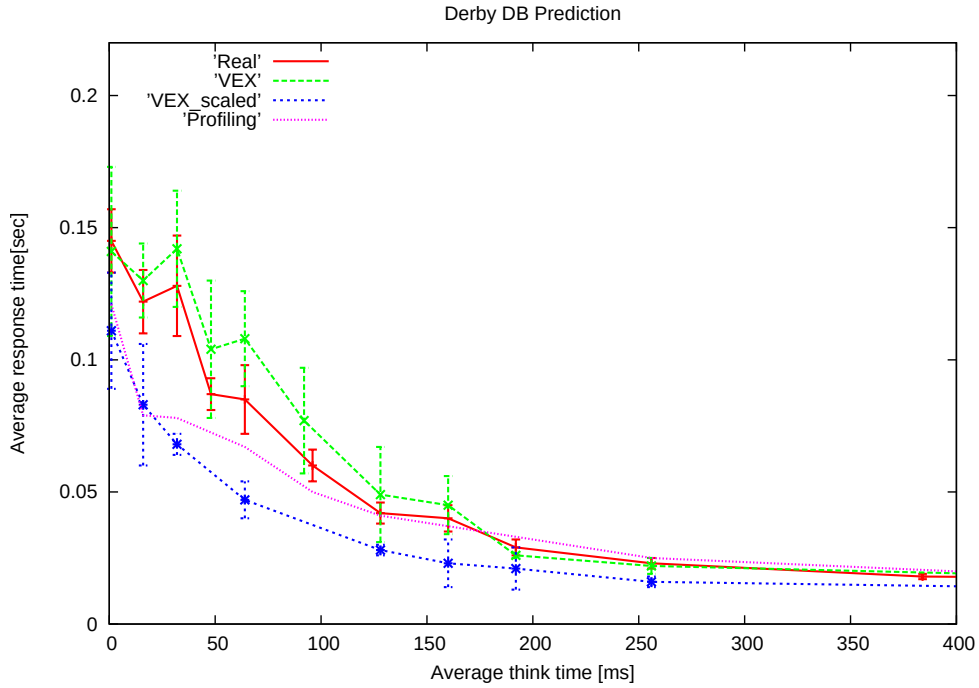


Figure 4.10: Derby DB results comparing the observed average response time to the VEX prediction and the time acquired by manually scaling the results of a profiler to time-scaled VEX simulation. Means of 10 measurements with approximately 90% confidence intervals are shown

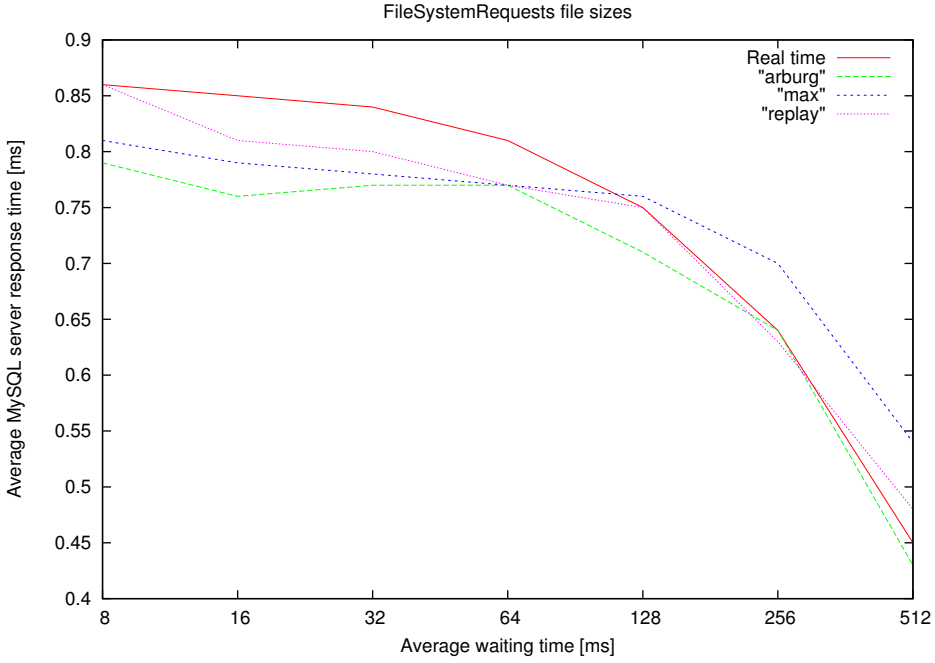


Figure 4.11: MySQL DB average response times for the real-time execution and the VEX simulation with the *arburg*, *max* and *replay* I/O prediction policies

Chapter 5

Java Instrumentation Environment

We introduce the Java Instrumentation Environment (JINE) which we developed to provide an interface to the VEX core for applications written in the Java programming language. Java was chosen because of its widespread use [127], the fact that many languages have been ported to the JVM, the well-documented dynamic bytecode instrumentation techniques, the user-friendly annotation capabilities and the availability of the JVMTI functionalities that support native-level callbacks upon the occurrence of system- or thread-related events. There is also the potential to develop an Integrated Development Environment (IDE) module for VEX, to facilitate its uptake from the software engineering community.

On the downside, Java adds another level of complexity, as it executes bytecode at the JVM-level and includes language features like JIT compilation and garbage collection. In this chapter, we present the design and implementation of JINE and elaborate on the language-specific challenges that arise and on the ways we have dealt with them.

The key contribution of this chapter is to demonstrate how to develop an instrumentation interface for the VEX core presented in Chapter 3. Explaining the requirements of VEX by the upper instrumentation layer, whilst elaborating on implementation choices made for JINE, this chapter can be used as a reference for the development of similar high-level interfaces for other programming languages. Particular focus is set on Java-related issues, which have become major challenges in our project, thus highlighting how non-trivial language-specific aspects may

affect the virtual execution framework. In an attempt to overcome these issues, we investigate the effect of tightly coupling VEX with the JVM, by directly modifying the latter's source code.

5.1 Design and implementation

JINE is built on top of the VEX core as illustrated in Figure 5.1. The lower-level of JINE is a JVMTI-agent, which is a shared library, loaded during the initialisation of the JVM. The JVMTI-agent is written in C++ and implements callbacks, that are automatically triggered upon predefined JVM events. The agent also exports to the upper JINE level a set of JNI (Java Native Interface) functions that invoke the VEX API. The higher-level JINE is a Java-agent, which dynamically checks each class during loading and transforms the methods of interest to call the exported JNI functions of the JVMTI-agent.

By employing bytecode instrumentation, JVMTI and JNI, the JINE system (JVMTI- and Java-agent):

- initialises and terminates the VEX simulator
- monitors thread state changes and acknowledges them to the VEX scheduling module
- supports the VEX profiling module that identifies regular methods, I/O methods and virtual specifications.

5.1.1 System initialisation and termination

Initialization

The JVMTI-agent of JINE is responsible for initialising the VEX core. This happens with the `onLoad` method, which is a callback that is always executed before any other JVM action. The VEX initialisation procedure is invoked, which spawns the VEX scheduler thread and allocates the data structures required for the simulation, based on the parameters provided by the users

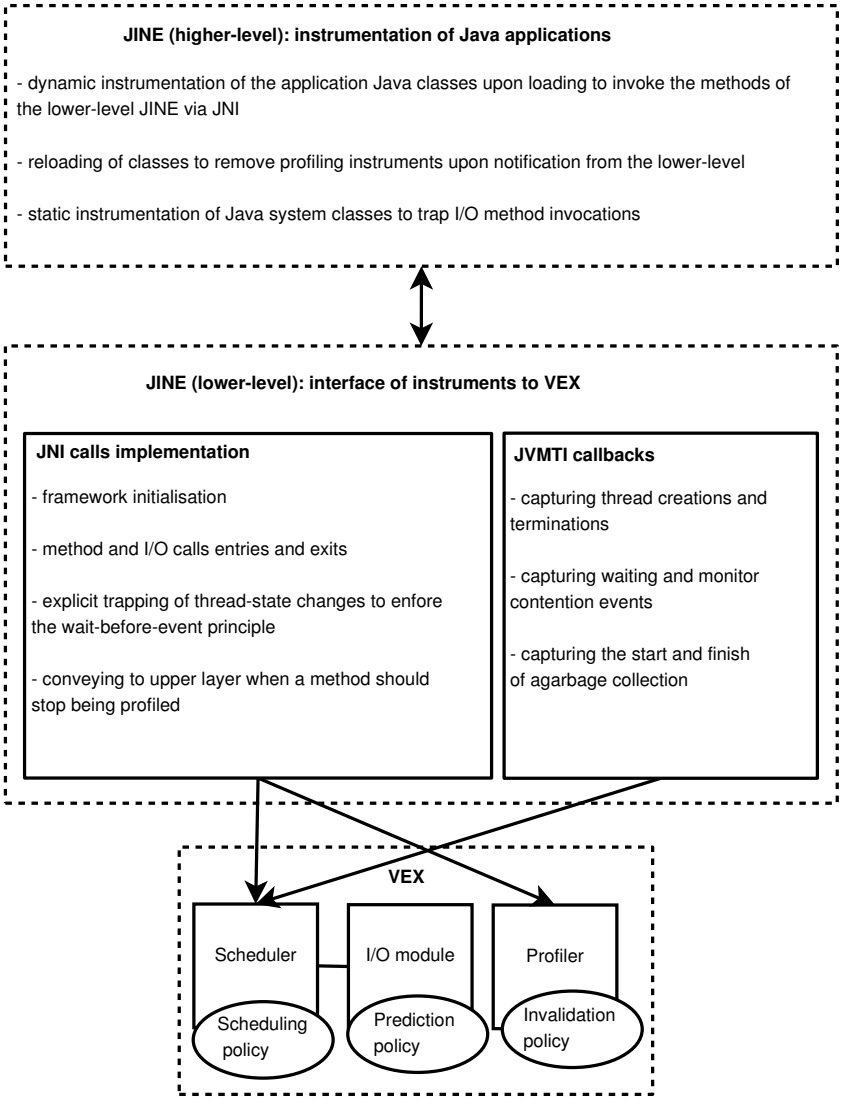


Figure 5.1: JINE design on top of VEX

when the JVM is launched from the command-line. A full list of VEX parameters is found in Appendix C. This approach ensures that the VEX scheduler and data structures are defined before any thread invokes one of their methods. The `onLoad` method registers all other JVMTI-agent callbacks that are executed at runtime, when the corresponding JVMTI-captured events occur.

To initialise the framework in absence of a JVMTI-similar feature, other languages may define a global static flag *initialised* or a condition depending on the initialisation of a VEX data structure, and include in the beginning of each instrumented code section. The first thread that finds the flag equal to *false* or finds that the data structure has not yet been initialised, is the one to initialise the framework.

Once loaded, and if explicitly defined by the user, the Java-agent of JINE supports the initialisation process, by first profiling the instrumentation overhead of the simulation host (see Section 3.2.3). By performing this measurement on the Java-agent level, JINE also includes the cost of invoking the JNI function that invokes the VEX method-logging call. To avoid warmup effects, the measurements are taken following an initial set of un-profiled ones. These results are placed in a file for future reference and thus this profiling overhead is paid only once per simulation host.

Termination

The JVMTI-agent offers a `VMDeath` callback that is automatically invoked when the JVM terminates. This method is registered in the `onLoad` method of the JVMTI-agent and calls the VEX termination procedure, which generates the various simulation outputs (see Section 3.2.4) and cleans up any data structures used.

For programming languages that don't support such a callback, VEX can be made to terminate when the "main" method exits, if this event can be identified or otherwise when the stack trace of the "main" method becomes empty. Though more generic than the use of a language callback, this approach assumes identification and different handling of the "main" thread.

5.1.2 Thread monitoring

The virtual time execution relies on the VEX scheduler being aware of any changes in the state of each thread, in order to control their progress accordingly. We review how the related state changes are monitored and the corresponding actions taken by VEX.

Creation

When a thread is created, it is registered to the VEX scheduler's data structures and pushed into the runnable threads queue. If set accordingly during the loading of the JVM, the JVMTI

callback `ThreadStart` is automatically invoked by the runtime, every time a thread is spawned during the "Live" phase. The "main" thread is an exception, as it is registered from the `VMInit` callback. To match language-specific thread ids to VEX threads, the instrumentation system assigns a unique thread id to every newly registered thread. JINE uses the id of each Java thread (the value of the *id* field from the `java.lang.Thread` class), to enable future mapping of Java thread ids to VEX threads.

JINE does not take into account JVM system threads (see Section 5.2.6) or threads from methods that are not instrumented. JINE supports this distinction by forcing each profiled thread that is about to spawn a new one, to enter its id in a special-purpose registry together with the virtual timestamp VT of its spawning time. Once a thread is created, it only puts itself under VEX's control if its id is in that registry. The initial timestamp VT is used to determine, when this thread is scheduled by VEX for the first time. This procedure is enabled via static instrumentation on the `java.lang.Thread` class.

All invocations to the private `java.lang.Thread/start0` method are also statically instrumented, to call JVMTI-agent methods before and after the creation of each thread. These methods call the corresponding VEX methods, which denote the passage of time during the new thread creation and increase a counter that describes the number of threads that are currently under creation. This counter is used to forbid "virtual leaps": if there is at least one pending spawning thread, then no timed-waiting thread can be resumed in virtual time (see corresponding challenge in Section 5.2.4). When a new thread is successfully registered in the queue of runnable threads, the counter is decremented.

Termination

The `ThreadEnd` JVMTI callback is registered to be activated, when a thread terminates execution. At that point the JVMTI-agent invokes the thread terminating function of VEX, which marks the timestamp of the event, updates the global virtual time (GVT) and releases the thread from the control of the VEX scheduler. Since the profiled virtual method times are maintained in thread-local data structures throughout a thread's execution, we also need to

update the globally aggregated method time results. The thread local data structures that are used during simulation are then cleaned up and the thread exits.

We note here that languages that do not support a similar feature as JVMTI callbacks on thread termination, can trigger this effect when the stack trace of a thread becomes empty (i.e. has no frames).

Sleeping

Sleeping durations are reflected in virtual times, in that a thread "wakes-up" only when the corresponding amount of time has elapsed in virtual time. A thread that is about to sleep, is pushed into the priority queue of runnable threads with a virtual timeout, that describes when it should resume. To trigger this handling, all invocations of the `sleep` method of the `java.lang.Thread` class are trapped into VEX. Interestingly, the JVM runtime does not allow us to statically instrument the `sleep0` private native method of that class (that leads a thread to sleep within the JVM), as it performs a test that checks whether the `java.lang.Thread` class defines a particular set of native methods: creating a `__vtf_native_prefix_vex_sleep0` method that invokes the original `sleep0`, as typically done in our native method handling technique (see Section 5.1.4 for details), is thus not possible. This means that JINE has to trap all calls to this method externally.

Dynamically instrumenting all invocations to `sleep` methods belonging to the `java.lang.Thread` class does not cover the overriding of this method from one of its subclasses. Identifying this set of `java.lang.Thread` subclasses S_{thr} is possible, but we don't generally know if a class S_i belongs to S_{thr}, when we are instrumenting the invocation of the `sleep` method of S_i in a random class C. Only if S_i has been loaded prior to C, are we able to know that S_i is a subclass of S_{thr} and thus instrument the `sleep` invocation accordingly. A solution to this problem would be to force the class loader to load S_{thr}, directly after finishing the loading of class C and then reloading C and instrumenting the `sleep` method accordingly, namely depending on whether S_{thr} was found to be a subclass of `java.lang.Thread` or not. As the complexity of the solution outweighs its potential benefits, we leave it as future work and treat all encoun-

tered `sleep` methods with one or two long integers as arguments as being from subclasses of `java.lang.Thread`, unless they belong to the same class C that invoked them.

Timed-waiting

VEX's handling of timed-waiting is very similar to sleeping, in that it sets a virtual timeout for its calling thread and pushes it back to the runnables queue. In contrast to the `sleep` method, the dynamic instrumentation of the `wait` call is simpler, because all classes are subclasses of the `java.lang.Object` class that defines it. This allows JINE to instrument all `wait` methods with one or two long integer arguments as virtual timeouts.

The key difference to the `sleep` method is that timed-waiting threads can be signalled, by `notify` or `notifyAll` calls. To accomplish this, JINE passes a unique object id to the VEX core, which is later matched with the ids of objects, that invoke `notify` or `notifyAll`. Threads block at a VEX conditional variable to wait for the virtual-timeout expiry (or signal arrival). Crucially, JINE needs to exit the monitor of the waiting object, to allow other threads to enter it, while the waiting thread is blocked within VEX. A JNI call is used to exit the monitor for as many times, N, as it had been entered before (synchronised blocks are re-entrant in Java). Similarly, on a virtual timeout or signal arrival, the monitor is re-entered N times, so that it can be correctly exited at the Java bytecode level, after the instrumentation code returns. To identify N at the instrumentation code layer, we iteratively release the monitor via a JNI call, until an exception it thrown. We catch the exception within the low-level JINE and set N equal to the times the monitor was released minus 1 (the release that led to the exception should not be counted).

Recall that, by handling the synchronisation of threads internally, VEX enforces the "update-before-event" principle (see Section 3.1.3), which ensures that state-changes are reflected to the system's data structures, before they are realised by the JVM or any other runtime. This ensures that the VEX scheduler can distinguish between blocked timed-waiting threads and timed-waiting threads that have been notified, but are still in the "Timed-waiting" state, because the state change has not been yet acknowledged to VEX.

Waiting

By default, JINE uses the JVMTI callbacks for `MonitorWait` and `MonitorWaited` events; these are triggered before and after a thread invokes the `Object.wait()` method without a timeout. The `MonitorContendedEnter` and `MonitorContendedEntered` callbacks, are respectively executed, when a thread has to block, upon entering a monitor already owned by another thread, and right after it is resumed and enters the previously suspended monitor.

Although these techniques simplify JINE monitor handling and are used by default, they are language-specific and are not consistent with the "update-before-event" approach enforced elsewhere in VEX. Indeed, suppose that a contended monitor is exited by thread T_B and thread T_A is blocked in the "Waiting" state trying to acquire it. Suppose also that T_B then becomes "Waiting" or "Timed-waiting" for some other reason, before the `MonitorContendedEntered` callback is triggered for T_A. From the perspective of the VEX scheduler and until T_A changes its state, when the code of the `MonitorContendedEntered` callback is invoked, both threads T_A and T_B are in the "Waiting" state. This allows it to schedule the next thread T_C, whose virtual timestamp might be ahead of T_A, thus violating the VEX scheduling policy. In the default JINE run, we accept the possibility of this kind of violation, taking care to limit the amount of time that T_C is allowed to leap forward in virtual time (see Section 5.2.4).

To overcome this issue and support languages without JVMTI-similar callbacks, JINE supports explicit monitor trapping: all monitor entries and exits and conditional waiting (Object waits) are notified to VEX via instrumentation. This allows VEX to manage thread synchronisation internally, in a way that adheres to the "update-before-event" principle. In this case `Object.wait()` calls are handled like timed-waiting calls, without a virtual timeout. They register themselves to the list of threads waiting on an *object id* and wait to be notified as explained in the following section. To pass the object ids to the VEX core, all `MONITORENTER` and `MONITOREXIT` bytecodes of all simulated methods are dynamically instrumented. Each synchronised method is also wrapped by a new method, that marks the monitor entering and exiting event, before and after invoking the original method. This approach shifts the responsibility of identifying lock contention to the VEX level. VEX's API includes the corresponding

`onRequestingLock` and `onReleaseLock` methods, that should be invoked before an entry to or an exit from a monitor. The immediate effect of calling `onReleaseLock` is to change the state of one of the threads, contending for entry to the monitor. This approach is potentially more generic and complies with the "update-before-event" principle, but carries a significant overhead as every monitor-related bytecode is trapped to the VEX layer. Language-specific features like lock re-entrance, that allows a thread to re-enter an already owned monitor, are handled with a special flag: if the flag is set then re-entrance is not allowed and threads can block on the locks they are holding (like in POSIX `mutex`). The explicit monitor trapping feature is nevertheless still at an experimental stage and has not been thoroughly tested.

Signalling

Object signalling methods like `notify` and `notifyAll` are dynamically instrumented, in the same way as the `Object.wait(J)` and `Object.wait(JJ)` methods. They interrupt waiting or timed-waiting threads in virtual time, by forcing the VEX scheduler to unblock them and set their virtual timestamp to the current GVT. To select the threads to receive the notification signal, the unique id of the notified Object is passed to VEX, which in turn matches it with all threads waiting on that id. Using that id `notify` and `notifyAll` a single random waiting thread is interrupted. In the case of a `notifyAll` call, all waiting threads do so.

As we mentioned in the previous section, JINE supports two ways of managing indefinite waiting with `Object.wait()`: using JVMTI callbacks or explicitly trapping the call and registering the id of the monitor that the thread will be waiting on, in the registry of waiting objects. The management of notifications works independently of the approach selected: if a record exists in the registry of waiting objects (corresponding to a timed- or indefinitely waiting thread), then the signal resumes the thread of that record. If no such record exists, or a `notifyAll` method is used, then JINE invokes `Object.notifyAll` at the JVM level, thus delegating the responsibility of resuming a blocked thread to the JVM. In turn, any resumed thread will trigger the `MonitorWaited` JVMTI callback, thus conveying the thread state change to VEX.

Threads are also interrupted upon invocation of the `interrupt` method of the `java.lang.Thread`

class. JINE uses static instrumentation of this class to trap such invocations and trigger a virtual time interrupt of a waiting, timed-waiting or sleeping thread. To interrupt timed-waiting and sleeping threads that are blocked within VEX, JINE obtains the originally assigned thread id (Java thread id) of the thread to be interrupted and provides it to VEX. In turn, the core finds the VEX thread matching that id and interrupts it in virtual time. If it is in the "Timed-waiting" (i.e. timed-waiting or sleeping) state or the "Waiting" state the interrupt is handled internally by VEX. Otherwise, if the thread is in the JVM-handled "Waiting" state, then the `interrupt` method is called at the Java level, after returning from the VEX instrumented code. JINE then relies on the JVM runtime to interrupt the waiting thread and trigger the `MonitorWaited` JVMTI callback, that eventually changes the thread state in VEX.

Performing I/O

Threads that perform I/O operations are trapped by instrumentation of the Java classes that invoke them. Analysing the hierarchy of the `java.io` package and according to [123], we identify the `java.io.FileInputStream`, `java.io.FileOutputStream` and `java.io.RandomAccessFile` as the classes whose native methods are invoked by all other java classes to request I/O. JINE employes static instrumentation in order to load modified versions of these I/O performing classes before the standard system classes do so by the *BootClassLoader*. The classes are preloaded with the *Xbootclasspath/p* command-line parameter and used throughout the VEX simulated run.

The effect of using the pre-instrumented `java.io` classes is that all their invocations (even by un-instrumented methods or threads) are trapped into VEX. To distinguish between profiled and unprofiled threads, JINE extends the `java.lang.Thread` class with a `vexInvocationPoint` integer field, initially set to zero. This field is changed to a unique non-zero value each time a statically instrumented I/O method is invoked by a method profiled by VEX. If *vexInvocationPoint* is zero, then the thread is not profiled and the I/O operation takes place normally. In the lower-level (VEX I/O module), this id can be used to distinguish between I/O invocation points. Crucially, this value plays the role of uniquely identifying the I/O point. This allows us

to group measurements and predict durations of I/O operations in a very fine-grained fashion, as elaborated in Chapter 4.

Our instrumentation approach covers only methods of the `java.io` package, thus omitting the available NIO (New I/O) operations. These methods perform asynchronous I/O, while multiplexing various I/O sources under the same call. For example, the `epoll_wait` method blocks a thread until an I/O event occurs on a predefined file descriptor. The duration of the I/O operation cannot be measured in real time as the observed time between the method entry and exit, because it would include the arbitrarily long blocking time of the thread. Moreover, the thread is not performing an I/O, but remains blocked until an I/O event occurs.

In this case, JINE explicitly sets the state of the thread as "Waiting", before it enters the call and as "Suspended" after it does so. The same handling is enforced for the `accept` system call, or the locking primitives of the `java.util.concurrency` package. We note here that this essentially constitutes a performance optimisation: if we had not explicitly instrumented these calls, VEX would eventually identify them as "Native waiting" (see Section 5.2.2) and disregard them, without knowing when they would resume. Explicitly trapping the entry and exit times alleviates this problem.

Yielding

Invoking the `yield` method of the `java.lang.Thread` class, results in calling the corresponding method of the VEX subsystem, that sets the currently running thread to "Runnable" and resumes it once all other runnable threads have executed once. Each invocation of the `yield` method in a VEX-simulated class is dynamically instrumented. This is tantamount to a `Thread.sleep`, that cannot be statically instrumented. This is because in order to instrument a native method M within a class, we need to replace it with a *non-native* method M to wrap a new native method named "_vex_M" that will be linked by the JVMTI agent to the native code of M (see Section 5.1.4). This violates the precondition of the HotSpot JVM, that requires that system classes define a predefined set of native methods (including M, which it replaced by its non-native wrapper).

Assuming that all classes defining `sleep` with one or two long integer arguments are subclasses of `java.lang.Thread`, all its invocations are treated as virtual timeouts. In contrast, `yield` methods are treated as such, only if they belong to the `java.lang.Thread` class. This is a design decision that mirrors the importance of trapping virtual timeouts from `sleep` calls and possibly allowing the miss of a yielding call (from a subclass of `java.lang.Thread`).

5.1.3 Method handling

The Java-agent dynamically instruments Java classes on loading, while parsing the virtual specifications. The method handling techniques concern method instrumentations, specifications parsing and adaptive profiling enforcement.

Method profiling

The Java-agent is responsible for registering methods, when it first loads the class containing them. Specifically, it assigns a method-id based on the `hashCode` of the Fully Qualified Name (FQN) of the method and registers the <method-id, FQN> pair to the VEX subsystem. We prefer this approach, because it yields reproducible mappings, that are useful for testing and debugging purposes. If such a feature is not necessary (e.g. in a production version of JINE), then sequentially assigning ids, which depends on the class loading sequence, works as well.

JINE adds instruments in the "beginning" and "end" of each method that use the generated method-id as an argument. The "beginning" of a method is defined to be the point before its first bytecode instruction. The "end" of a method is either the last bytecode before any kind of method returning instruction (i.e. `*RETURN`) or before an exception-throwing bytecode, like `ATHROW`. To avoid treating exceptions caught within a method as method-exit points, JINE uses a counter for opened exception-catching blocks: only if this counter is zero, is `ATHROW` considered a method exit point.

By default JINE instruments all application methods. To selectively profile only a subset of them, users may provide a file with method FQNs. Note here that by "application methods",

we exclude system classes methods, like the ones found in the `java` or `sun` packages. By instrumenting such methods, we would inadvertently increase the overhead of the framework, without any real benefit in the performance results provided. If any such method is the bottleneck, then its performance overhead is still reflected in the execution times of its calling method.

Our design decision to disregard execution times of system classes, is also reflected in our choice not to trigger JVMTI callbacks on method entry and exit. Although the usage of JVMTI callbacks does not require any bytecode instrumentation effort, it is less flexible and efficient than our approach: JINE decides exactly which methods it wants to instrument and this filtering occurs only once during class loading. This is quite different from a filtering scheme at the JVMTI-agent level, where a filtered-out method would trigger the callback, only to return, but after paying the price for transforming the Java strings that constitute the FQN of the executed method to a method id.

Virtual specifications

The virtual specifications are defined as method annotations. For example, time-scaling factors are annotated in the form: `@Accelerate(speedup =` TSF`)`, for $TSF > 0$. If $TSF > 1$ then we have a speedup and if $0 < TSF < 1$ we have a slowdown in the virtual execution time of that method. ASM's `AnnotationVisitor` interface is used to parse the annotations, when the class of the annotated method is loaded. The parsed TSF is registered with the method-id of the annotated method to VEX. Every time the method with this method-id is entered, the corresponding TSF is going to be applied for the entire thread execution, until the method is exited. If no annotation is defined for a method, then its regular execution time remains unmodified during the virtual time accounting.

Adaptive profiling support

JINE supports the adaptive profiling approach of VEX (see Section 3.2.3), by checking VEX's profiling invalidation policy from the JVMTI-agent library every time a method exits. When the

invalidation policy determines that a method M should no longer be profiled, the JVMTI-agent invokes the "profiling invalidation enforcer" module. This uses JNI to invoke a Java method from the native level, passing to it the method-id of M. In turn the high-level method reloads and redefines the class that contains M, instrumenting the method entry and exit points of all non-invalidated methods (thus not M). However, instruments regarding thread state changes are still added to all methods. To disregard the methods of the class that have already been invalidated, the JINE high-level module, maintains an array of all invalidated method-ids for each class. Synchronising on the JVMTI-agent level, JINE forces only one thread to invoke the class reloading and invalidation scheme at a time.

Subsequent invocations of non-profiled methods just invoke its method body, i.e. without the entry and exit instruments. The execution times of the method up to the invalidation point are still output in the profile, but the durations of any subsequent calls to the method are reflected in the execution times of its calling method.

Frameworks that do not support class reloading or dynamic instrumentation may use a separate boolean registry to distinguish methods that need to be profiled from the ones that do not.

5.1.4 Instrumentation techniques

All class transformations and parsing of annotations described above are offered by the ASM [22] framework for modifying and creating Java bytecode. VEX uses the *core* API of ASM, which represents Java classes as a sequence of events and not as objects (as in the *tree* API). Each event represents an element of the class, such as its header, a field, a method declaration, an instruction, etc. The event-based API defines the set of possible events and the order in which they must occur. A class parser generates one event per element that is parsed, allowing JINE to add its instrumentation code for each parsed event, according to whether it is a method-entry, an exiting bytecode instruction, an invocation to a method of particular interest (`Object.wait`, `Thread.sleep`) etc. At the end of the bytecode instrumentation, a class writer generates compiled classes from sequences of such events.

Dynamic instrumentation

The instrumentation layer of JINE is defined as a Java-agent that is loaded before the invocation of the `main` method of the running class. It implements a `ClassFileTransformer` interface that transforms the bytecode of classes, as they are being loaded by the JVM.

Instruments are added to the original bytecode, according to the needs of JINE and the user-defined runtime parameters, like the definitions of packages, classes or methods that should be profiled. The "limitProfiling=none" parameter allows the exclusion of all profiling instruments, apart from the ones on the "main" method. Classes excluded from profiling instrumentation like this, only have the invocations to methods that denote thread-state changes instrumented.

Static instrumentation

System classes that are loaded before the Java-agent by the `SystemClassLoader`, cannot be reloaded, instrumented and re-used by the `AppClassLoader` at runtime, which is lower in the hierarchy of JVM class-loaders. Therefore, the only way to modify their behaviour is to explicitly reload them at runtime, transform them and output them into a class file one at a time. All these instrumented system classes are put into a *jar* library, that is preloaded with the *-Xbootclasspath/p* JVM command-line parameter. The JVM loads these classes prior to the original system classes from `rt.jar`. At the point where the latter would be normally loaded, the JVM disregards them as already loaded classes.

Native method instrumentation

Native methods do not have any bytecode to which we can add the VEX requested features. For this reason, whenever we want to instrument a native method M_n, we create a new method with exactly the same Fully-Qualified Name (FQN) M_n, but without the "native" modifier. We define the bytecode of this new method according to the instrumentation needs, invoking at some point a method with the same FQN prefixed with "`__vtf_native_prefix_`". We then define the method `__vtf_native_prefix_`M_n with the "native" identifier. Clearly, all original

calls to M_n invoke the new Java method, which in turn invokes the new native method. As we still need to link to the correct native implementation of method M_n, when executing `__vtf_native_prefix_`M_n, we use the `SetNativeMethodPrefix` JVMTI call, during JVMTI-agent loading.

5.1.5 Usage

From the user perspective, the framework achieves transparency by requiring a minimal set of actions. Users may annotate their methods with the virtual specifications of their choice, that determine the time-scaling factors of each method. If the source code is not available, a file with FQNs and time-scaling factors can be provided instead. This approach requires absolutely no changes to the source code of the application under performance testing. Selection of the methods to be instrumented can also be parameterised with the help of a similar FQN file. Note that the main method is always selected by default.

Having set the virtual specifications and denoted the classes to be profiled, users may use the VEX framework on any Java application. This is allowed by defining JINE's native shared library as the application's JVMTI-agent and JINE's transformer class as its Java-agent, using the *agentpath* and *javaagent* command-line parameters respectively. The statically instrumented system classes also need to be defined as preloaded classes to "override" the loading of the regular I/O classes' implementations with the *Xbootclasspath/p* parameter. More information about the usage and parameters of the JINE/VEX framework are found in Appendix C.

If no parameters are defined, then the Java application is executed in virtual time, having all its methods profiled, without any adaptive profiling scheme.

5.1.6 Summary

Table 5.1 summarises the key features of the interface between VEX and the upper instrumentation layer. Each row shows the event that needs to be instrumented, the action it triggers in

the virtual execution core and how it is handled by JINE. To facilitate implementation for other languages, we show in the last column of the table how Java-specific features, like JVMTI, can be reproduced in a generic way. Bytecode instrumentation approaches that are used in JINE need to be replaced by language-specific instrumentation frameworks. Dynamic and static instrumentation techniques could be used in any way that delivers the requested information to VEX, irrespective of the way that this happens in JINE. For example, the Pin [111] framework identifies system calls and I/O invocation points at runtime and can thus dynamically instrument such calls, despite of the fact that JINE utilises static techniques to do so.

Event	**Action**	**JINE framework**	**Generic framework**
Program startup	VEX initialization	JVMTI onLoad	During first invocation to VEX
Program exit	VEX output and cleanup	JVMTI VMDeath	"Main" method or thread exiting
Thread create	Register thread to VEX	JVMTI ThreadStart	Register at any invocation of an unregistered thread
Thread terminate	Merge thread with global results	JVMTI ThreadEnd	Upon a method exit that leaves an empty thread stack trace
Thread sleep	Virtual timeout	Dynamic instrumentation of all invocations to any sleep(J) and sleep(JJ) methods	
Thread timed-waiting	Virtual timeout (notifiable)	Dynamic instrumentation of all invocations to any wait(J) and wait(JJ) methods	
Thread wait	Thread blocked in VEX	JVMTI MonitorWait(ed) and MonitorContendedEnter(ed)	Instrument all thread locks and waiting calls and pass a lock id to VEX
Thread signaling	Interrupt a waiting thread	Dynamic instrumentation of all invocations to notify(J) and notifyAll(JJ) methods	
Thread interrupt	Send interrupt signal	Static instrumentation of Thread.interrupt method	
Thread I/O	Issue I/O in virtual time	Static instrumentation of native I/O methods of core java.io classes	
Thread explicit wait	Set and unset a thread waiting	Dynamic instrumentation of invocations to a set of predefined methods	
Thread yield	Yield in virtual time	Dynamic instrumentation of all invocations to Thread.yield()	
Method entry and exit	Log method entry and exit times	Dynamic instrumentation before method's first instruction and before returning or throwing exception	
Virtual specs	Register specifications to VEX	Annotation parsing	File with virtual specs to methods
Adaptive profiling	Stop profiling a method	Reload method class and remove instruments	Use method-id to boolean registry to invalidate profiling

Table 5.1: Reference table on how JINE interfaces with VEX

5.2 Challenges in Java virtual time execution

Most of the problems we encountered during this thesis were found during the verification and validation of the VEX and JINE integrated framework. VEX is in theory independent of the programming language that the application is written in, as long as all events described in Table 5.1 are conveyed to it. In case of Java applications, some of these events take place in native code within the JVM (internal synchronisation points) and are not forwarded to VEX. As a result the combination of VEX and JINE can lead to deadlocks, high simulation overheads and high prediction errors. To address these issues, we typically had to amend some aspect of VEX or JINE with special handling code. In this section we present these challenges and our approaches in dealing with them.

The assumptions at this point regard that each Java thread corresponds to exactly one system thread. This allows the VEX scheduler to control each thread by sending POSIX signals to its lightweight process id. This also means that "green-threads", which are logical threads on the JVM-level that are mapped to system threads on a many-to-one basis, are not allowed. We also assume that the JVM handles POSIX signals, either by not ignoring them or by using a signal-chaining facility to resolve conflicts between JVM and VEX signals.

5.2.1 Implementation note: Deadlocks in JVM

The fact that internal JVM synchronisation points are not forwarded to VEX has led to the identification of a problem during the implementation of lower-level JINE. Specifically, in order to invoke Java methods from the JVMTI-agent code, lower-level JINE uses JNI calls. This happens in three cases: when a thread is spawned, when a class is reloaded to invalidate the profiling of a method and when a thread exits and enters an object monitor, before and after being set into the "Timed-waiting" state. In each case, the JVMTI-agent executes code that leads back into the JVM runtime. A deadlock can then occur when the JNI method is invoked, if another thread that has been suspended by the VEX scheduler whilst holding a JVM-internal lock.

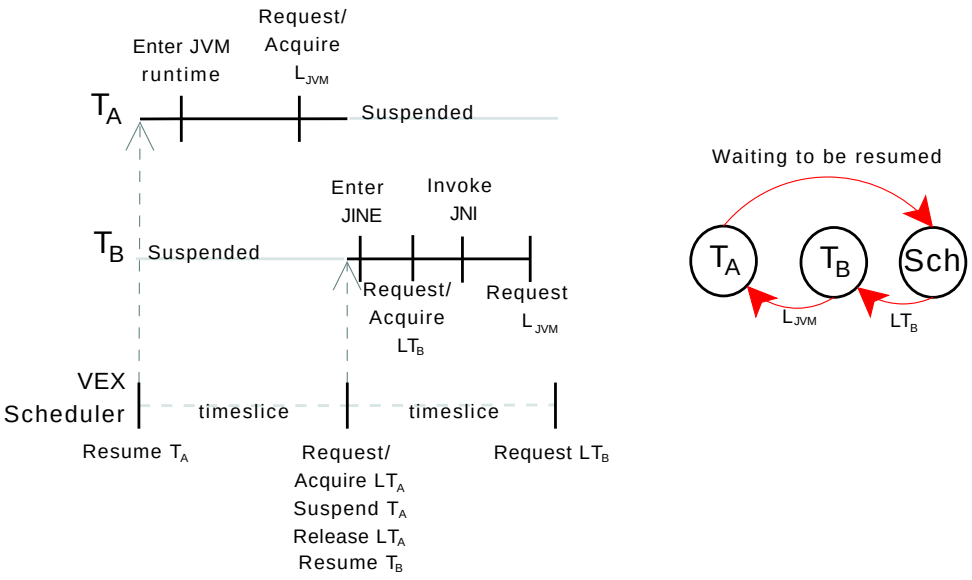


Figure 5.2: Deadlock as a result of VEX scheduling and JNI invocation from JINE

To explain how the deadlock occurs, recall that threads that execute VEX instrumented code hold a lock that forbids the VEX scheduler from suspending them. If the timeslice of the thread expires in the meantime, the VEX scheduler is forced to block on this lock until the thread releases it, i.e. leaves the instrumentation code. Crucially, a running thread should never block, while holding this lock, as this would block the VEX scheduler and in turn deadlock the simulation. Although abiding by this rule for our own code is feasible, we cannot guarantee this when performing JNI invocations that run code within the JVM.

We illustrate the problem in Figure 5.2. Thread T_A has been suspended by VEX, before releasing a JVM-internal lock L_{JVM}. While T_A remains suspended, thread T_B acquires its VEX lock LT_B to forbid the scheduler from suspending it and invokes a JNI call, which for a JVM-internal reason needs to acquire L_{JVM}. T_B blocks inside the JVM and when its timeslice expires, the VEX scheduler will block on LT_B, while trying to suspend T_B. This leads to a deadlock, because T_A cannot be resumed by the VEX scheduler to release L_{JVM}.

To avoid this problem, threads that invoke a JNI call from instrumented code should never hold their VEX lock while doing so. This allows the VEX scheduler to suspend them, while they are executing the code of the instruments added by our simulator. Threads are not normally allowed to do so, because the virtual time update within the VEX signal handler would wrongly include overheads from the execution of the simulation framework. By updating virtual timestamps,

before releasing their VEX lock, this overhead can be minimised.

We note here that there is nothing wrong with threads being suspended whilst holding a JVM lock. However, in this case the progress of the threads may be somehow perturbed compared to the real time execution: eventually the VEX scheduler resumes the thread that is holding the JVM lock and the virtual time execution will continue as normal.

5.2.2 Native waiting threads

VEX works under the assumption that all thread state changes are conveyed to the framework's scheduler, which then responds to them in such a way as to generate a valid thread interleaving in virtual time. By using a variety of JNI, JVMTI and instrumentation techniques, JINE generally accomplishes this objective. A key exception concerns thread state changes that take place in native code other than VEX (e.g. a native library used via JNI) or within the JVM runtime. For example, Sun's HotSpot virtual machine synchronises Java threads with a mechanism called "safepointing": operations like the stop-the-world part of a garbage collection prerequisite that all threads are idle. This JVM internal synchronisation mechanism happens asynchronously and is not exported via JVMTI to the agent; such events are not, and cannot be, instrumented to be trapped by any of the techniques used by JINE. We call threads blocked in a native method, "Native-waiting" threads.

The overall effect of "Native-waiting" is that the VEX scheduler falsely considers a blocked thread as running. Accordingly, it allows entire timeslices to be wasted, increasing the overhead of the simulation without any benefit. The problem is exacerbated by the fact that the scheduler suspends a thread whose timeslice has expired, only if its CPU-time has progressed adequately (75% of timeslice duration), since the last time the thread was resumed. This enforces a fair schedule virtual execution, in cases where the OS scheduler does not allocate enough CPU time to a VEX running thread during its timeslice (for example due to background load). Suppose now that a thread is actually "Native-waiting": its CPU-time is essentially never increased as the thread is blocked (to be precise there is a slight nanosecond-level increase, due to the overhead from the execution of the VEX signal handler that asks threads to suspend). In

turn, the VEX scheduler is not suspending the thread and the simulation remains in this state "forever", leading to a type of livelock, which renders the simulation approach impractical.

To deal with this issue, we extend the VEX core with a new "Native-waiting" state and a mechanism to identify such threads. Two types of identification heuristics are supported:

- The *counter-based* method uses a flag to determine whether the thread is executing VEX code and counters to find how many times the thread has tried to acquire a VEX lock (and could have been blocked waiting for it). If the counters are zero, then we calculate the real time and the virtual time durations since the last time the thread was resumed; if the virtual time duration is less than p percent of real time (here $p = 1\%$), then the we infer that the thread is not making any CPU progress and therefore should be considered as *Native-Waiting*.

- The *stack-trace based* method uses the `libunwind` library [34] to parse the thread's stack trace, searching for an invocation of the `wait` method of the `pthread` library.

To avoid the overhead of unrolling the stack in the stack-trace based method, the two methods can be combined to the *hybrid* method: only if the counter-based method identifies a thread as possibly *Native-Waiting* will the stack-trace based method be used to verify this.

Once identified, "Native-waiting" threads become "uncontrolled", in that they are no longer scheduled by VEX. It is up to the JVM and the OS scheduler to resume them when they become unblocked. Note that the presence of "uncontrolled" threads creates additional load to the OS scheduler which, under the VEX paradigm for a uniprocessor simulation, should have only one application thread to run at each point. Nevertheless, "uncontrolled" threads still execute instrumented code and trigger JVMTI callbacks. We take advantage of this, and modify all VEX instruments, to immediately suspend any "uncontrolled" threads executing them, resetting them under VEX's control.

An alternative approach to identifying whether threads are still blocked in native code is the "poll_nw" one. According to the "poll_nw" approach, the VEX scheduler polls all "Native-

waiting" threads after each timeslice and monitors their progress in CPU time. Even microsecond-level values of CPU time progress are considered enough to denote that the thread is running again, since the CPU time progress of a blocked thread is zero and is only increased by the execution of the VEX signal handler that is triggered after a polling message from the VEX scheduler. The disadvantage of polling lies in the high number of POSIX signals that might be possibly sent to threads that spend most of their time waiting to perform an action (see Section 5.2.6). The VEX core is extended to offer this functionality, by maintaining a priority queue of native waiting threads and polling the thread with the lowest virtual timestamp, to determine whether it is still native waiting.

VEX also uses a Linux-specific approach to identifying "Native waiting" threads, by parsing the *status* file under the */proc/* $<$ *pid* $>$ directory, where the kernel logs the state of each process. A thread that is found to be "Runnable" (R) can be scheduled by the OS scheduler, which means that it is not blocked in native code. Therefore a thread in this state should not be perceived as native waiting by VEX. "Sleeping" (S) or "Stopped" (T) threads should.

Native waiting threads also affect simulation accuracy. The CPU-time T_{nw} from the time they change their state to "Native-waiting" T_0 to when they return under VEX's control T_{end} is still measured correctly, but this duration is mapped onto the virtual timeline at the point when the previously "uncontrolled" thread re-enters VEX. This can lead to arbitrarily long CPU-time progress being mapped onto the virtual timeline at a random point in time, thus increasing the predicted execution time of all methods executed at that point. In practice, the existence of various traps to VEX in the program in combination with the polling methods for prompt identification of "Native-waiting" threads, suffice to limit this effect. Our work presented in [10] follows this way of accounting the time of native-waiting threads. VEX then adds this duration $t = T_{end} - T_0$ to the virtual timeline, moving from GVT_0 to GVT_1. It also keeps the amount of time added like this in a register (called *unknownParallelTime*), to overlap it with any time progress that has happened by normally scheduled threads between GVT_0 and GVT_1.

To summarise, "Native-waiting" threads are a nuisance, because they increase the overhead of the framework and are not controlled by the VEX scheduler. The latter has two consequences:

- The CPU-time progress of "Native-waiting" threads in only mapped on the virtual timeline when they return under VEX's control.

- The VEX scheduler is not aware of the exact state of a native-waiting thread, which might affect scheduling decisions related to leaps forward in virtual time (see corresponding paragraph in this section).

5.2.3 Safepoint synchronisation

Safepoints are used in the HotSpot JVM to synchronise all threads, before a virtual machine operation like garbage collection, class redefinition, locking bias revocation etc. occurs. In JDK6, the bytecode interpreter checks a safepoint-specific flag, before allowing threads to make progress. Otherwise, it blocks them on a safepoint-related condition variable. Threads that execute compiled bytecode (native methods) have the pages of the compiled instructions read protected via `mprotect`, which leads them to raise a segmentation fault signal when their next instruction is read. The signal is then caught by the JVM, forcing such threads to block within the signal handler. Threads that are executing JNI code are taken care of by checking, when a thread returns to the JVM from native code. Blocked threads within the JVM or newly spawned threads that are not executing yet are also similarly handled with JVM internal flags. But under which of these categories do the VEX-suspended threads fall?

Unfortunately, none! Regardless of whether they were executing interpreted or compiled bytecode, they are currently blocked within the signal handler of VEX; from JVM's perspective they are normally runnable threads. When the JVM handles a safepoint synchronisation request for virtual machine operation VM_{op}, VEX-suspended threads have to explicitly block at the safepoint condition variable, before VM_{op} takes place. JVM expects them to either invoke the next bytecode instruction or cause a memory protection fault; until they do, the safepoint requesting thread (typically the *VMThread* that processes VM operations from a queue), will be spinning or yielding the processor to allow them to make progress.

Now assume a virtual execution that, at the point when the safepoint flag is raised and the

safepoint synchronisation is about to take place, has one running thread T_1 and $(N-1)$ runnable (suspended by VEX) threads $T_2, \ldots, T_N$. T_1 is immediately trapped into the safepoint condition variable, but without VEX knowing about this. Only when the timeslice of the thread expires will VEX use the lack of CPU-time progress or the stack-trace to determine that the thread is native waiting, set it to "uncontrolled" and schedule the next runnable thread (let it be T_2). When T_2 executes it will also be trapped at the safepoint condition variable, which is again only recognised when its timeslice expires. Each of the N threads becomes "Native-waiting" one-by-one, after time $N * s$, where s the VEX scheduler timeslice. This significantly increases the overhead of the framework compared to the case, when the change of the thread's state at the safepoint condition variable would be directly conveyed to VEX. In the meantime the $VMThread$ that issued the safepoint synchronisation request is spinning or yielding within a loop (that counts threads that have still not reported themselves to be blocked in a safepoint), thus wasting CPU cycles. Upon completion of the VM operation, all N threads will become runnable again, allowed to make progress concurrently on the single core, violating the strict virtual time execution as "uncontrolled" native-waiting threads. They return under VEX's control in the first time they invoke one of its instruments, at which point they map the entire CPU-time progress from when they became native-waiting onto the virtual timeline.

This is the penalty to pay for making VEX JVM-agnostic and blocking threads with signals. In Chapter 6 we see how limiting the VEX scheduler timeslice can contribute to reducing the overheads from this effect.

5.2.4 Leaps forward in virtual time

In this section we address the issue of time warps [69, 14], in which periods of inactivity can be skipped by moving directly to the next event in virtual time. In the context of VEX and JINE, time warps take place when a thread T_{tw} with timestamp t_{now} starts waiting with a timeout t or sleeping for an amount of time t. VEX updates its timestamp to $t_{expire} = t_{now} + t$ and pushes it into the priority queue of runnable threads. This thread is resumed when it reaches the front of the priority queue, i.e. when t_{expire} is the lowest timestamp among all *runnable*

threads. At that point we say that the thread has performed a "virtual leap forward" (VLF) to t_{expire}. It is important to appreciate that the real time duration until t_{expire} is irrelevant; what matters is the number of events that have to be processed until that point in virtual time. For example, if no events (i.e. threads in a runnable state) are scheduled to occur until t_{expire}, then the virtual leap forward may be performed immediately, regardless of how long it would normally need to reach the timeout in real time.

The problem is that VLFs work correctly, as long as *non-runnable* threads with a smaller timestamp than t_{expire} cannot change their state in the meantime. This is true in discrete-event simulation [47], where (omitting parallel DES synchronisation issues) all events up to t_{expire} have already been processed: as no further changes can take place, leaping forward to t_{expire} is valid. In VEX, however, events that take place at a lower-level (within the JVM or the OS) are not trapped into our framework and can change the state of a thread without VEX knowing it. If such an event takes place at t_{ev}, $t_{now} < t_{ev} < t_{expire}$, and VEX leaps forward in virtual time before processing it, then the code following t_{ev} will only be executed in virtual time *after* t_{expire}. This is not the behaviour that would be exhibited in the correct virtual time schedule, where t_{ev} happens *before* t_{expire}.

We refer to this problem as a *premature* VLF. Premature VLFs that are incurred by lower-level events that deal with thread creation and termination have been solved by trapping code that will eventually cause them to VEX. For example, a counter C, that denotes the number of threads under creation, is increased at t_0 *before* a thread starting method is invoked at the JVM level. C is only decreased at t_1 *after* the thread has registered itself to VEX. Without this handling, premature VLFs could occur between t_0 and t_1, when the new thread is not yet visible to VEX. Allowing VLFs only when $C = 0$ solves this problem, at the cost of additional instrumentation code whenever a thread is created.

Another case where premature leaps are avoided, due to special purpose extension to VEX, concerns joining threads. When a thread T_x is about to terminate its execution, it triggers the corresponding JVMTI callback and enters VEX to update its status and relinquish itself from the control of the scheduler. At that point T_x becomes "invisible" to the scheduler, which

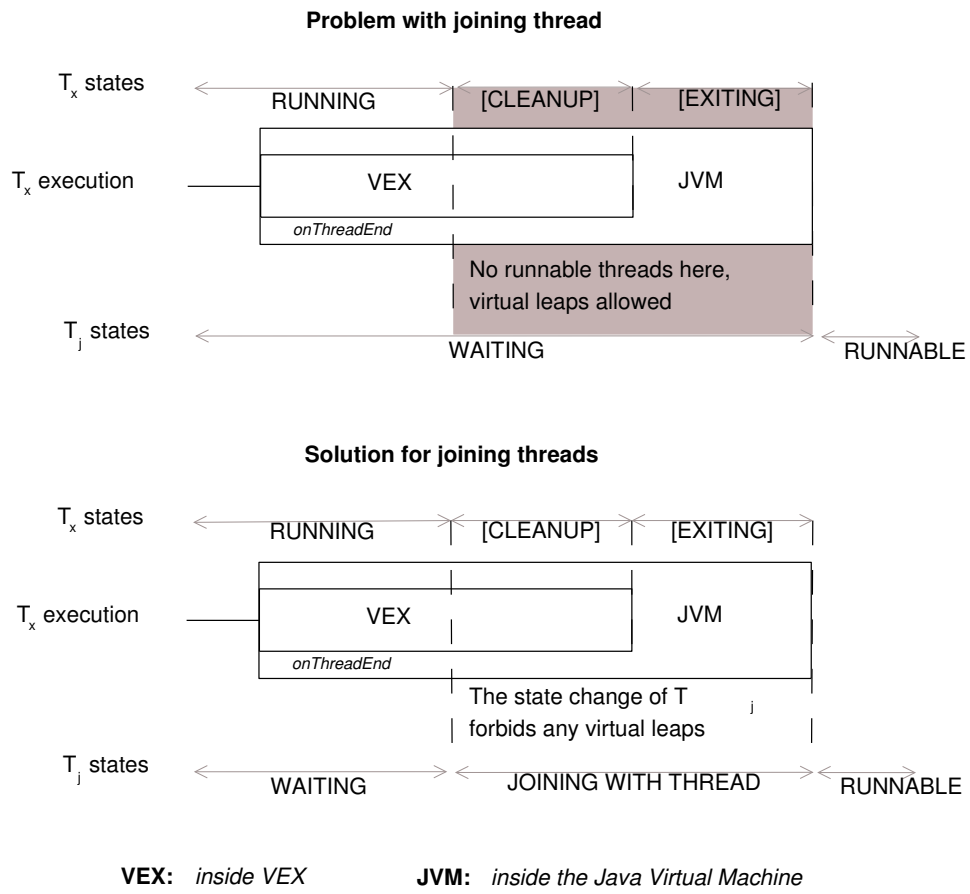


Figure 5.3: Handling of joining threads in VEX to avoid virtual time leaps

resumes a new thread. In the meantime, the exiting thread T_x cleans up its data structures and merges its VEX profiling results with the global registry, before terminating within the JVM. Now imagine that only two other threads exist in the simulation: one (T_j) that is waiting to join with T_x (i.e. waiting for T_x to terminate) and another one T_l that is waiting to leap forward in virtual time. At the moment that T_x logs its exit in VEX, the only runnable thread from the perspective of the simulator is T_l, as T_j is still indefinitely blocked (shaded part of upper figure in Figure 5.3). T_j will only become runnable once T_x terminates. The problem is that in the meantime, since no runnable thread exists in the simulator, VEX allows premature VLFs. JINE solves this problem, by monitoring the joining of threads: every thread T_j that waits on another thread T_x to exit, registers itself into a data structure, using the id of T_x as key. When T_x exits VEX, it checks this registry for any waiting threads and updates their VEX state, before the JVM actually does so, thus forbidding any premature VLFs (see Figure 5.3).

5.2.5 Heuristics for countering premature VLFs

Avoiding premature VLFs by indirectly monitoring system events, as happens with thread creation and termination events, is not possible in other cases. We elaborate on them in this section.

Case 1: A blocking thread is becoming runnable

Blocked threads do not interfere with the decision of the VEX scheduler on whether to allow a VLF or not: as they are not able to make any progress, the VEX scheduler can allow a VLF, without worrying whether it violates the correct order of virtual execution. However, VEX is not able to distinguish between "definitely" blocked threads and blocked threads that are allowed to resume (i.e. the monitor they have been waiting to acquire has been released). For example, imagine a thread T that has been blocked, waiting to enter a contended monitor. There exists some small but inevitable delay from the time that the contended monitor is freed to the time that the corresponding JVMTI callback is issued to convey this event to VEX. During this delay and though T is about to resume execution, the VEX scheduler considers it blocked and may thus allow leaps forward in virtual time.

Case 2: Native-waiting threads

Native waiting threads (see Section 5.2.2) work by definition at a lower-level than VEX and become "uncontrolled" only to re-enter the simulator when they invoke instrumented code. In the meantime, their execution can progress, despite the fact that they are perceived as native waiting by VEX. In this case the forward leap should not be allowed. Polling the current state of these threads is possible by signalling them to acquire and parse their current stack traces, by comparing the progress of their CPU time counters since their last state change, or even checking the status of the corresponding system lightweight process. Knowing their state, VEX can decide on whether they are currently blocked or not. However, the current state of a thread does not provide any knowledge for any imminent change: what if the currently blocked thread

on a safepoint synchronisation is about to resume because the safepoint was released?

One might argue that, since native-waiting is related to JVM internal synchronisation (for instance safepoint synchronisations), then the scheduler should never allow any virtual leaps forward, until the internal synchronisation is resolved. Unfortunately, this could deadlock the system if some of the threads T_{after} that need to synchronise on the safepoints are runnable, but suspended by VEX at a virtual timestamp after t_{expire}. Threads T_{after} are not resumed by the VEX scheduler before the virtual leap forward occurs, the virtual leap is not allowed, whilst other threads $T_{nw} \neq T_{after}$ are native waiting, and the native waiting T_{nw} threads will never change their state until the T_{after} ones synchronise on the safepoint. This is clearly a deadlock. Even if no such T_{after} threads exist, the "Native waiting" state covers, on top of JVM internal synchronisation, all blocking behaviours not trapped by VEX. These are:

- A thread is synchronised at a native library.
- A thread invokes an *accept* system call on a socket file descriptor, blocks until a thread connects to the socket (unless marked as non-blocking).
- A thread is set to block via the classes of the *java.concurrency* package, that do not convey this to VEX via JVMTI.
- A thread performs a blocking I/O operation in an uninstrumented method. In this case, the I/O is not handled by VEX, which in turn leads to the blocking not being handled by VEX.
- A thread performs other timeout-based system calls, like *poll*, *select* or *epoll_wait* with a long timeout.

This means that we cannot make any assumptions that the native waiting state will have to change before t_{expire}.

Case 3: An I/O operation is taking place

Threads that perform I/O operations in the learning phase are removed from the queue of runnable threads, to allow other threads to make progress in parallel (see Chapter 4). Typically such threads are going to complete the I/O operation at some point and if this point is before t_{expire}, any virtual leap forward should be forbidden in light of such operations. But what if these operations are about to block in I/O? For instance, in the case of socket I/O, imagine a thread T_{send} sending a message to a socket that a thread T_{recv} is blocked indefinitely waiting for an I/O, and then T_{send} becoming blocked itself for some reason. Eventually, the message arrives and T_{recv} resumes, which means that the leap to t_{expire} should not take place in the meantime. On the other hand, if the message was sent to a different address, then the thread should remain blocked and there would be no reason to forbid the virtual leap forward (in fact doing so might lead to a deadlock). One approach is to monitor all I/O traffic between the application threads. However, this complicates the extension to a distributed version of VEX, since we are not able to determine the outcome of each socket I/O operation.

The "Waiting", "Native-waiting" and "I/O" thread problems demonstrate that, at the current level that VEX is operating, it is not always possible to know whether a state change will occur before t_{expire}. The only definite solution in these cases is to monitor the status of each thread at the OS level. This is something we do not wish to do at this point, as one of the main design objectives of VEX has been to remain at the user-level (see Section 9.3.1 on future work). One partial solution, which retains the advantage that no change is required to the OS, is to use heuristics.

The heuristics we have chosen for this purpose are designed to be very conservative. This is because, during our initial verification efforts of VEX and JINE, we found that a single premature VLF can have significant effects on the prediction accuracy. For example, the SPECjvm2008 benchmarks (evaluated in Section 6.4) have a timeout thread that determines the duration that each benchmark is allowed to iterate over its methods. If the timeout thread is wrongly allowed to leap forward in virtual time, then there is a danger that the thread will cause the benchmark to finish prematurely. The conservatism of our heuristics comes at the

cost of simulation speed and at an additional effort to guarantee that a livelock will never occur: we should not mistakenly forbid a VLF forever.

Heuristic 1: Timestamp

If the t_{expire} timestamp is lower than the GVT then the leap is always allowed.

Heuristic 2: Thread states

We examine the set of threads T_L with a virtual timestamp lower than t_{expire}. We forbid leaps forward if there exist threads in T_L which are in the:

- "Running" or "Suspended" state, which either occurs if a previously I/O-issuing thread has changed its state or in multicore simulations (see Chapter 7).
- "Performing I/O" state, performing non socket-related I/O, which is assumed to never block.
- "Native-waiting" state and there exist no runnable threads with timestamps higher than t_{expire}. If such threads existed, we would assume that the "Native-waiting" thread is trying to synchronise with the runnable threads at a safepoint and would allow the forward leap.

If these checks do not suffice to forbid the leap forward and there still exist threads in T_L that are in any of the ambiguous states ("Waiting", "Native waiting" and "I/O"), then we defer the decision to *Rule 3*. We note here that threads in the "Waiting" state are only taken into account if their timestamp VT is $VT > GVT - \Delta t$. This heuristic limits the number of "Waiting" threads that affect the forward leap decision, by excluding the ones that have been blocked for a "sufficiently long" time Δt. It was specifically introduced to disregard threads that wait to join with a set of worker threads for the entire duration of an application. Δt has been selected to be equal to twice the duration of the leap forward.

Heuristic 3: Time counters

The next heuristic that we employ is to identify changes in the GVT and the thread-local time counters between consecutive timeslices. If their values are different, then there is some activity in the simulated program and the leap forward should not be allowed. Although this heuristic covers any activity in the system, it might end up wasting scheduler timeslices, if no progress is made by any thread.

Rule 4: System states

After two consecutive timeslices without any GVT or thread-local virtual time progress, VEX polls the *status* file under the $/proc/ < pid >$ directory to check the OS states of all threads in ambiguous states (prior to t_{expire}). If all of them are found in either "stopped", "sleeping" or "uninterruptible sleep" states, then the VLF is allowed, no matter how far ahead in virtual time it might be.

Conclusion

Premature VLFs are caused by the lack of exact knowledge about the states of VEX-controlled threads in the OS and the JVM. We introduce a number of heuristics to avoid premature VLFs, based on the GVT, the states of the threads before the timestamp of the VLF, the virtual time progress between two consecutive VEX scheduler timeslices and the OS system thread states. All of these have to be evaluated to allow a VLF to take place in the case that some threads are in a potentially changing state (see previous section). This renders our approach conservative, but also costly, due to the fact that scheduler timeslices can be wasted, whilst trying to identify whether something changed in the simulation. If no such changes occur, then the leap will eventually be allowed.

An alternative solution, besides modifying the OS kernel or using heuristics, is to use partial information on state changes in the JVM. For example, we could trap safepoint synchronisations and notify VEX immediately after such operations finished. Threads which would otherwise

be in the "Native-waiting" state would then become runnable without having to enter another JINE instrument. This would limit the time that a premature VLF could occur. Although this does not resolve this problem completely, preliminary results suggest that such an approach has a positive effect on the VEX simulation. Further exploration of this idea is left as future work (see Section 9.3.1).

5.2.6 Java system threads

Operations like garbage collection or just-in-time (JIT) compilation are performed by special-purpose internal JVM threads. They are spawned during the initialisation of the virtual machine and perform their operations periodically or when some condition is satisfied. In this section we present the problems related to handling such Java system threads in VEX.

The main problem with internal JVM threads is that they are not handled by JINE techniques: they never enter any method that is instrumented by VEX, they interact with other VM threads within the JVM and most of them do not even trigger the JVMTI ThreadStart callback, as they are created before the "Live" phase, when the callback is enabled. As system threads are typically waiting on a conditional variable when they are not actively performing their tasks, their state from VEX's perspective is "Native-waiting".

Controlling threads which are mostly in the "Native-waiting" state is hard, because the scheduler is never certain whether they are blocked or resumed. "Native-waiting" threads are hidden from the simulation and are only re-scheduled by VEX once they invoke one of the methods instrumented by VEX or its thread-event callbacks (see Section 5.2.2). Since none of these takes place for Java system threads, we cannot apply the existing mechanism for handling "Native-waiting" threads.

An alternative approach could be to poll Java system threads continuously, to identify whether they are still waiting or have made any progress. This idea is both inefficient and ineffective: excluding the case that a thread is actually running, a large number of signals to the "Native-waiting" system threads is required just to find out that the thread has made some progress at

some point in the past.

In either case, controlling system threads in VEX would introduce a simulation performance penalty. Being unaware of the low-level interaction of system threads with Java application threads, VEX assigns a full timeslice to each of them (system or application thread) and identifies the "Native-waiting" state only at the end of the timeslice. Adding the CPU-time progress of threads that are mostly "Native waiting" is also error prone in terms of prediction accuracy, as the mapping of this duration to the global virtual timeline happens at an arbitrary point.

We have decided to disregard system threads, because we cannot effectively handle them in VEX and they can become sources of significant overhead and misprediction. Excluding GC-threads and assuming a long-running Java application, their performance effect is typically low. As for GC system threads, their performance impact is taken into account with a JVMTI callback that is triggered in the beginning and end of each collection (see Section 5.2.8).

5.2.7 JIT compilation

Just-in-time (JIT) compilation is a standard feature of JVM implementations. By generating native code and subsequently executing it instead of interpreting bytecodes, the JIT compiler achieves significant speedups compared to the interpreted mode. The time, place and optimisation techniques of the compilation typically depend on a number of heuristics, based on method invocations, existence of loops etc; this comprises the "adaptive optimisation" aspect of a JVM, where only the most performance critical parts of the bytecode are compiled to native code.

The VEX and JINE virtual time framework handles compiled bytecode perfectly well, profiling it as it would for any other method. Indeed, the framework is agnostic as to whether the executed method is compiled or not. Nevertheless, the addition of the JINE instruments on the bytecode-level changes the body of the methods to be JIT-compiled, leading to a different native code than the one of the uninstrumented run. The mere execution of the VEX and JINE instruments affects decisions on when and where JIT compilation should take place. For

short-running programs this can lead to a discrepancy between the JIT-compilation in the real and virtual time execution.

Longer running programs (i.e. applications where the total running time is significantly higher than the startup time) are generally expected to be less affected by JIT compilation effects [51]. In particular for VEX and JINE, we can assume that in the steady-state the set of JIT compiled methods of the virtual time execution, will closely match the ones from the real time execution. The effect of the JINE instrumentation on the JIT compilation is also alleviated in the steady-state. This is because JINE's adaptive profiling (see Section 5.1.3) removes the instruments added to the most frequently executed methods. After the classes that contain these methods have been reloaded without the VEX profiling instruments and assuming that no thread state changing instruments exist in such methods, then the JIT compiler will generate the same code as in the real time execution.

5.2.8 Garbage collection

Profiling the duration of garbage collection is performed in real time within JVMTI callbacks. These are triggered before the beginning and after the end of the pause phase of the garbage collection. Although the JVMTI reference [129] states that some collectors may never generate these events, all types of collectors in the HotSpot JVM (stop-the-world, concurrent mark-sweep and incremental) do so. It is the frequency and duration of the pause phase that change amongst different collectors. The callback before the beginning of the collection forces the VEX scheduler to suspend the currently running thread. It then measures the duration of the collection in real time and adds the observed time to the GVT. Threads are resumed after the collection at the new value of GVT. This allows the garbage collection duration to be reflected in the virtual times of the methods that were executed when the collection started.

The VEX simulator works at the native level and only uses a minimal set of constant JNI global references (for adaptive profiling enforcement purposes), which are never garbage collected. JINE affects the garbage collection only during class loading, where a number of objects that support dynamic instrumentation are created. Such objects only have a short-term effect, as

they die soon after their creation in the young generation. A similar effect is also triggered during the *profiling invalidation* process, that creates method adaptor objects that instrument reloaded methods (see Section 5.1.3).

Chapter 6

Results

In this chapter we evaluate VEX and JINE on contrived test-cases and single- and multi-threaded CPU-bound and generic tests of small and medium size. The main objective is to verify that the predictions without any time-scaling match the observed running times of the tests. This would mean that the effects of enforcing a virtual schedule, adding instruments and measuring mainly CPU- and partially real time durations reproduce in virtual time a sufficiently accurate profile of the original behaviour of each benchmark. In addition, for selected cases we were able to evalute time-scaling results, either via a theoretical model or via manual change of the application code. Each experiment was run on one of Hosts 1-4 from Appendix A with their second core disabled. This will be mentioned separately for each test.

6.1 M/M/1 simulation

In order to explore the prediction accuracy and execution time overheads for CPU-bound applications, we have written a Java program that emulates a simple client-server system with a single-threaded server. Clients are modelled by a single thread that sleeps for a random *think* time and then spawns a new thread that adds a request to a server queue. A single server thread retrieves jobs from this queue and models a service time by making a random number of iterations of a predefined loop; the server thread thus does 'actual' work in order to emulate

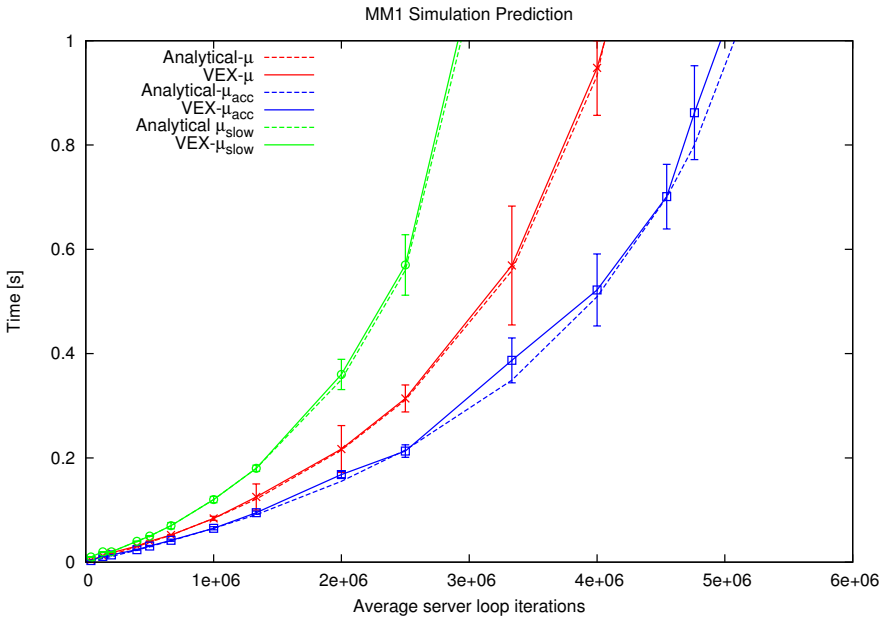


Figure 6.1: M/M/1 simulation results with approximately 90% confidence intervals

individual service times.

By arranging the think times and service times to be exponentially distributed, with equivalent arrival and service rates λ and μ respectively, the expected response time of the server can be calculated straightforwardly from the mean waiting time in an M/M/1 queue, vi.z $R = \frac{1}{\mu - \lambda}$. In order to test the framework's ability to reproduce the expected behaviour we fix λ and increase the number of iterations of the server loop to model a decreasing service rate. This imposes an increasing load on the server. For a particular service rate, we then explore two ways of scaling the performance of the server: 1. By changing the number of iterations of the server loop and 2. By using the method scaling factor of the virtual time framework to scale the measured execution times for the server code in virtual time. In both cases we compare the results with those expected from M/M/1 queueing theory.

Figure 6.1 shows the results obtained for $\lambda = 400$ and with two scaling factors, viz. a 20% increase in service rate ("μ_{acc}" in Figure 6.1) and a 33% decrease in service rate ("μ_{slow}" in Figure 6.1) on Host-4 (see Appendix A). Note that the choice of $\lambda = 400$ has been chosen simply to reduce the time taken to perform the experiments. To cater for the small request inter-arrival times used in the experiment (2.5ms average) we set the VEX scheduler's timeslice s to 1ms. Using the default timeslice of 100ms would result in artificially lower observed waiting

times. For example, if $0 \leq C \leq s$, is the time between a new thread T being spawned and the VEX scheduler resuming it, then T will be inserted into the queue C time units later than it should be. This results in up to s lower queueing times per request. Limiting the scheduler timeslice we reduce this effect.

The observed response times match well the predicted times for both manual scaling of the service times and those resulting from method scaling in virtual time. This is encouraging, although the application is very simple and incurs no I/O.

6.2 JGF single-threaded

Next we evaluate VEX on the sequential benchmarks developed by the Distributed and High Performance Computing group of the University of Adelaide [135], that have been later integrated into the sequential Java Grande Forum (JGF) suite. They consist of benchmarks that stress-test specific JVM operations and kernels of well-known CPU-intensive computation kernels. The profile of the classes and methods instrumented by JINE (i.e. excluding system classes) is presented in Table 6.1, while their descriptions are found in Appendix B. As only

Benchmark	**Classes**	**Meth.**	**Instr.**	**java.io method calls**	**Synchr. points**	**Methods / class**	**Instr. / method**	**Invalid. meth-ods**	**Recurs. meth-ods**	**VEX adapted %**
Array creation	6	33	1436	26	0	6	33	0	0	0.00
Exc. handling	7	35	1240	26	0	5	35	2	0	5.71
Generic	6	34	2641	26	0	6	34	1	0	2.94
Loop	6	33	1074	26	32	6	33	0	0	0.00
Method calling	6	43	1680	29	5	7	43	10	0	23.26
Object creation	16	43	2146	26	0	3	43	10	0	23.26
Thread	7	36	1030	27	0	5	36	0	0	0.00
EP	8	43	1158	29	0	5	43	1	0	2.33
FFT	7	50	1792	27	0	7	50	3	0	6.00
Fibonacci	7	40	931	26	0	6	40	0	1	2.50
Hanoi	7	41	1004	26	0	6	41	0	1	2.44
Sieve	7	40	1021	26	0	6	40	2	0	5.00
IS	8	50	1719	38	0	6	50	0	0	0.00

Table 6.1: Code profile for the single-threaded JGF benchmarks as returned by JINE

a single thread drives the workload and since JVM system threads are not profiled, the VEX scheduler is mostly idle, thus enabling us to isolate the effect of VEX instrumentation on the simulation error and overhead. We exclude the section-1 benchmarks of the suite that stress-test aspects of the JVM, like object generation, fork-join etc.; these are evaluated in the next section

(for multi-threaded JGF benchmarks). The section-2 kernels (from *EP* to *IS*) are executed in real time and their results are then compared with the ones acquired without time-scaling via VEX simulation. Each run is repeated three times on Host-1 (see Appendix A).

Threads: 1	**Real time**		**VEX full profiling**					**VEX adaptive profiling**				
Benchm.	**Res.**	**Time**	**M.inv.**	**Res.**	**Time**	**Err.%**	**O/h**	**M.inv.**	**Res.**	**Time**	**Err.%**	**O/h**
EP	2.65	2.74	33.5M	12.11	48.63	355.91	17.70	10040	6.21	6.53	133.98	2.38
FFT	1.31	1.39	4.2M	2.31	7.11	75.78	5.08	30050	1.24	2.56	-5.69	1.83
Fibonacci	0.60	0.68	204.7M	63.11	279.09	10367.33	409.22	10054	0.6	0.80	-0.50	1.17
Hanoi	0.20	0.28	50.3M	15.50	67.66	7461.46	235.75	10049	0.20	0.41	1.46	1.45
Sieve	0.21	0.27	25	0.17	0.35	-15.57	1.27	25	0.16	0.33	-20.75	1.21
IS	3.21	3.52	53.8M	17.29	76.14	437.92	21.58	20062	3.99	5.07	24.20	1.44
Geo-mean						492.22	24.89				8.08	1.53
Mean						3118.99	115.10				31.10	1.58

Table 6.2: Results for the single-threaded JGF. Means of 3 runs are shown for the full profile and 30 runs for the adaptive profiling mode

Threads: 1	**Real time -Xint**		**VEX 10k adaptive profiling -Xint**					**VEX 1M adaptive profiling -Xint**				
Benchm.	**Res.**	**Time**	**M.inv.**	**Res.**	**Time**	**Err.%**	**O/h**	**M.inv.**	**Res.**	**Time**	**Err.%**	**O/h**
EP	31.91	32.11	10040	32.11	34.82	0.62	1.08	1M	30.95	51.71	-3.03	1.61
FFT	5.77	5.78	30050	5.45	6.09	-5.46	1.05	2M	5.53	13.41	-4.16	2.32
Fibonacci	9.24	9.33	10054	8.82	9.75	-4.61	1.05	1M	9.16	16.71	-0.88	1.79
Hanoi	6.03	6.28	10049	5.34	5.91	-11.48	0.94	1M	5.57	10.68	-7.74	1.70
Sieve	2.45	2.31	25	2.16	2.52	-11.80	1.09	25	2.16	3.54	-11.71	1.53
IS	18.08	22.59	20062	16.52	21.13	-8.63	0.94	2M	17.21	39.03	-4.80	1.73
Geomean						5.13	1.02				4.11	1.76
Mean						7.10	1.03				5.39	1.78

Table 6.3: Results for the single-threaded JGF with the *-Xint* flag. Means of 3 runs are shown

The results of the VEX simulation are shown in Table 6.2. The predictions under the "VEX full" columns correspond to the case where all methods have profiling instruments added to their bodies. These exhibit high prediction errors and simulation overheads. Enabling the adaptive profiling scheme, we dynamically remove the instruments in methods that are invoked for more I times (here $I = 10,000$), thus limiting the observer effects caused by excessive profiling (see Section 3.3.2). The results under the "VEX adaptive" columns demonstrate a clear improvement over the fully instrumented bytecode with a geometric mean of the absolute prediction error being reduced to 8.08%.

Nevertheless, even in the adaptive profiling case the *EP* and *IS* present high prediction errors. We investigate the cause of the prediction error in *EP*, by using the "-XX:PrintAssembly" JVM option with the *hsdis* (HotSpot disassembly) library, that prints out the methods and generated assembly of the JIT-compiled bytecode. We find that the method `ep/KernelEP.ep()` is consistently compiled to native code in the Real Time Execution (RTE), but not in the Virtual Time one (VTE). Although this method is invoked only once, it includes a long loop

that performs the kernel's computation and is thus compiled using the on-stack replacement (OSR) technique of the HotSpot JVM that compiles methods that are already being executed on some thread's stack. As the counters that enable OSR (number of leaps backwards) remain the same in both executions, we can only assume that the reason that the JIT compiler "decides" not to compile the method in the VTE lies in the additional JINE-instrumentation related methods, that are compiled prior to `ep/KernelEP.ep()`. This increases the number of requests for method compilations from the application threads and might affect the decision for the OSR compilation of `ep/KernelEP.ep()` as a means to limit background compilation time. Iterating the kernel for a few more times enables the compilation of the method, thus showing that it is not the existence of JINE instruments that somehow hinders this in the long run.

Indeed, executing the benchmark in the JVM interpreted mode without any compilation (presented in Table 6.3 for $I = 10,000$ and $I = 1,000,000$) we acquire a much better prediction for both *EP* and *IS* kernels and overall prediction errors of between 4% and 5% . We expect that longer running applications that are not sensitive to such small scale warmup effects are not going to suffer from this issue, though we note that there exist no guarantees for the matching of JIT compilation patterns between the RTE and the VTE.

The conclusions from this set of short single-threaded benchmarks are:

- Full profiling can incur very high prediction errors and simulation penalties for short running benchmarks, even when no VEX scheduling is involved.

- Adaptive profiling is necessary to overcome this issue, by limiting the invocation rate of instrumented methods and invalidating the profiling of recursive methods.

- JIT compilation effects in short running programs may still yield high mispredictions regardless of the number of method invocations.

6.3 JGF multi-threaded

6.3.1 Basic results

The next set of results evaluate the prediction accuracy of VEX for the multi-threaded Java Grande Forum (JGF) benchmarks [68]. They comprise stress tests that measure the throughput of specified JVM operations and well-known CPU-bound computation kernels from cryptography (*Crypt*), linear algebra (*SparseMatmult*, *Series*, *LUFact*, *SOR*), simulation (*MonteCarlo*) and graphic processsing (*Moldyn*, *RayTracer*) (see Appendix B for more details). Again, we execute the benchmarks in real time and then in virtual time without time-scaling, in hope that the two measurements should match.

Table 6.4 presents the results for the JGF multi-threaded benchmarks for 8 threads on a uniprocessor (Host-1 in Appendix A) divided in four categories: one for the stress tests and one for each of the various workload sizes for each benchmark A, B and C (ranked from the smallest to the largest). The high prediction errors of the stress tests are attributed to the fact that events like spawning and joining threads, as well as invocations of `Thread.yield()`, are instrumented and trapped within VEX. The effect of executing instrumentation code at a high rate affects both the prediction accuracy and the simulation overhead, due to instrumentation intrusion effects [86], as we have already shown in Section 3.3.2 for profiling instruments. This applies to most stress tests, yielding an overall simulation overhead of 10.88 for the experiments undertaken. It also affects benchmarks like the *SimpleBarrier*, where the increased rate of entering the thread-state changing instruments increases the virtual time, leading to an underprediction of the expected benchmark throughput. This also applies to an extent for the *ForkJoin* benchmark, which, in contrast to *SimpleBarrier*, presents an overprediction. This is attributed to the fact that the thread creating system call *clone*, which is one of the main factors for the execution time of this benchmark, is not accounted for in CPU time. This leads to less virtual time being accounted for and thus a higher throughput prediction.

TournamentBarrier synchronises the 8 benchmark threads in a loop, by iteratively invoking the `Thread.yield()` method. When the yielding thread is resumed it checks if the barrier condition

has been fulfilled in the meantime; otherwise, it yields again. Recall (see Section 5.1.2) that calls to `Thread.yield()` are dynamically replaced by a custom VEX routine that parses the priority queue of runnable threads to find the maximum virtual timestamp, assigns it (plus 1ns) to the yielding thread and signals the VEX scheduler to make another thread runnable. As this process is iterated at a very high rate in the *TournamentBarrier* implementation, the running time of the VTE is much higher than that of the RTE, despite the significantly lower count of invocations to the yield call. In fact, we had to lower the number of iterations of the *TournamentBarrier* benchmark to an absolute minimum (200 iterations per thread), to acquire a result from VEX within a reasonable amount of simulation time. Despite the existence of a few accurately predicted benchmarks, the stress tests clearly expose the limitations of the current framework.

The virtual time measurements for the CPU-bound tests of the remaining three categories match better the corresponding real time results. As we would expect VEX may suffer from high prediction errors for shorter running benchmarks. For example, the prediction error of the *SparseMatMult* benchmark for the smallest workload is over 100% and for the largest only 1.81%. Similarly, *Crypt* errors range from -53% to -8% and taking the geometric mean of the absolute prediction errors shows a clear trend when increasing the workload size: both prediction errors (in absolute terms) and simulation overheads decrease. This is attributed to the warm-up effects, like the fact that selection of methods to be compiled by JIT might differ between the two short runs due to the existence of VEX methods or that the execution time of the compilation threads is not reflected in VEX results, as JVM system threads are not controlled by the VEX scheduler.

The three outliers that do not follow this pattern are the *LUFact*, *MolDyn* and *RayTracer* benchmarks. They all invoke `Thread.yield()` at a high rate in RTE (see corresponding column under Real Time in Table 6.4), but not quite so in VTE. *LUFact* calls the instrumented code that replaces `Thread.yield()` on average from 51,632 for workload A to 140,574 times for workload C, which incurs a high simulation overhead penalty (651x, 207x, 13x), but offers rather accurate predictions ranging from -0.36% to -15.53%. In contrast, the virtual execution of *MolDyn* only invokes yield 4,848 times on average, which limits the simulation overhead,

Threads: 8		Real time			VEX					
Ctg	**Benchmark**	**Result**	**Time**	**Yields**	**Meth.inv**	**Result**	**Time**	**Yields**	**Err.%**	**O/h**
Stress	SimpleBarrier	39098	3.94	0	1050014	20375	62.89	0	-47.89	15.96
Stress	TournamentBarrier	2899	0.17	71750	1644	6897	73.81	3410	137.91	444.65
Stress	MethodSync	3057480	25.63	0	45364	3146020	28.59	0	2.9	1.12
Stress	ObjectSync	2735510	25.63	0	45364	2850100	28.59	0	4.19	1.12
Stress	ForkJoin	1797	10.42	0	90057	3092	178.685	0	81.59	17.14
	Geomean-stress:								23.08	10.88
A	Crypt	0.48	0.67	0	82	0.22	0.46	0	-53.94	0.69
A	LUFact	0.28	0.44	155622	188482	0.28	181.79	51632	-0.36	651.58
A	Moldyn	3.98	4.45	2254349	403729	2.41	5.16	4849	-39.52	1.15
A	MonteCarlo	3.27	3.99	0	942576	2.86	14.01	0	-12.55	3.51
A	RayTracer	2.39	2.46	101900	904818	2.86	16.71	309	19.95	6.79
A	Series	8.55	8.58	0	30046	8.11	9.79	0	-5.07	1.14
A	SOR	8.7	9.87	0	48	9.92	13.12	0	13.99	1.32
A	SparseMatMult	0.38	0.7	0	47	0.84	1.27	0	122.87	1.8
	Geomean-A:								14.21	3.68
B	Crypt	1.69	1.89	0	82	1.59	2.05	0	-6.04	1.08
B	LUFact	1.54	1.73	513973	220239	1.37	284.75	96124	-11.27	207.85
B	Moldyn	28.08	27.99	1595165	456856	24.26	28.42	4848	-13.62	1.01
B	MonteCarlo	16.14	16.96	0	3393542	15.25	49.96	0	-5.51	2.94
B	RayTracer	19.63	19.99	333433	856716	29.08	34.65	72447	48.1	1.73
B	Series	86.73	86.63	0	88311	82.75	89.37	0	-4.6	1.03
B	SOR	9.23	11.38	0	48	10.1	13.61	0	9.36	1.19
B	SparseMatMult	0.65	1.03	0	47	0.9	1.41	0	38.56	1.36
	Geomean-B:								11.92	2.57
C	Crypt	3.84	4.24	0	82	3.52	4.81	0	-8.33	1.13
C	LUFact	11.27	11.66	1823172	257735	9.52	131.21	140574	-15.53	13.79
C	Series	1470.32	1471.6	0	2007761	1375.12	2061.2	0	-6.47	1.5
C	SOR	10.03	11.78	0	48	10.65	14.31	0	6.16	1.21
C	SparseMatMult	10.95	11.86	0	47	11.15	13.77	0	1.81	1.16
	Geomean-C:								6.22	2.01

Table 6.4: Results for the multi-threaded JGF benchmarks with 8 threads. The results of the stress tests are throughputs of events, while the rest are in seconds. The count of invocations to the `Thread.yield()` is shown to demonstrate the effect on prediction accuracy and simulation overhead. Means of 3 runs are shown.

but adds to the misprediction error, as the real time cost of yielding (making the `sched_yield` system call, context switching and waiting to be scheduled again) is only partially compensated. *RayTracer*'s problem is quite different, because it is the only benchmark that over-estimates the execution time of the RTE. In such cases, the problem lies typically in JIT compilation: using the "-XX:+PrintCompilation" flag we find that the `raytracer/RayTracer.render` method, which applies the main algorithm of the benchmark is not being compiled to native code by the JVM in the VTE. This creates a slower execution in virtual time, which increases the predicted execution time.

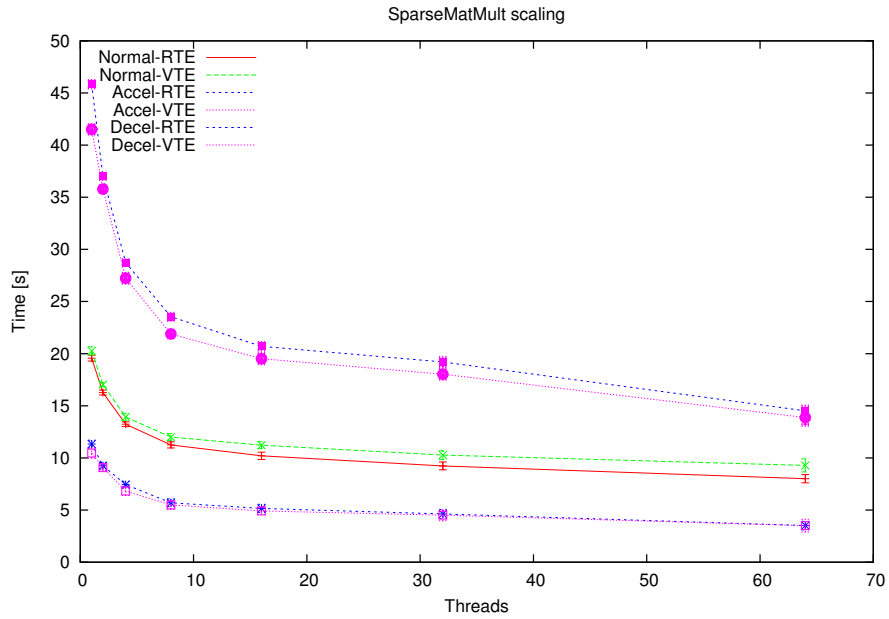


Figure 6.2: Thread and time scaling results for the *SparseMatMult* benchmark of the multi-threaded JGF suite. Means of 3 measurements per thread are shown with 95% confidence intervals

6.3.2 Thread and time scaling

We select the *SparseMatMult* benchmark using the largest workload C to investigate how the estimates of the VTE match the real times when we increase the number of concurrently running threads from 1 to 64. The results shown in Figure 6.2 under the "Normal" category show how the virtual time predicted times are close to the real ones. Interestingly, the time of execution is reduced in both cases as we increase the number of threads, which we attribute to better locality of reference for larger numbers of threads. Accessing the PAPI hardware performance counters, we verify that executions with 1 thread have 21% more L2 cache misses and 32% higher cycles per instruction (CPI) than the 32 thread case. It is important to note how these changes are reflected in the VTE as well.

Up to this point our validation approach consists of comparing real times with un-scaled virtual time predictions. Evaluating time-scaling is possible if there exists a way to acquire "expected" scaled times for the RTE, which typically can only be achieved via a theoretical model (like in the M/M/1 test). The *SparseMatMult* benchmark allows us to validate time scaling, because it iterates its entire kernel computation within a loop. Increasing or decreasing the iterations

of the loop enables us to simulate time-scaling in real time and then compare the acquired "scaled" real times with the VEX provided ones. Specifically, we double and then half the loop counts and acquire the scaled ("Accel" and "Decel") real execution times. Then we revert to the original code and annotate it to define a virtual time speedup and then a slowdown of a factor of 2. The results are also depicted in graph Figure 6.2, showing overall a very good prediction accuracy of 5.5% on average (4.0% geometric mean).

The conclusions from the set of multi-threaded JGF benchmarks are the following:

- Stress tests that involve thread creations and thread state changes at a high rate lead to increased prediction errors, due to instrumentation intrusion effects.

- The size of the simulated program matters: due to warm-up effects the same benchmarks performed much better with larger workloads.

- Invocations of the `Thread.yield()` method create prediction errors due to the fact that the duration of the system call *sched_yield* is not measured in CPU time and is compensated with a constant value and that the method's behaviour is simulated in virtual time taking into account only the VEX controlled threads.

- Differences in JIT compilation due to VEX can also be found in longer running programs like *RayTracer*.

- Thread and time scaling do not significantly affect the prediction accuracy of medium size CPU-bound kernels.

- System effects like increased cache misses and CPIs in the real time execution are also captured in the virtual time simulation.

6.4 SPECjvm2008

6.4.1 Single-threaded

The SPECjvm2008 benchmark suite [128], which has a larger and more varied code-base (Table 6.5 and see [124] for an analytical performance characterisation), is used to investigate VEX and JINE on production-level Java code. Each benchmark in SPECjvm2008 measures its performance in terms of a *throughput*, as measured by the number of benchmark iterations completed per unit time. By automatically replacing real with virtual execution times, VEX

Benchmark	Class	Meth.	Instr.	java. io meth. calls	Sync. points	Meth / class	Instr / meth.	Inval. meth.	Recurs. meth.	VEX adapt%
compiler.compiler	44	491	18042	499	16	11	37	16	1	3.46
compiler.sunflow	44	486	17990	497	16	11	37	16	1	3.50
compress	46	454	18002	435	16	10	40	17	0	3.74
crypto.aes	35	415	16819	431	16	12	41	0	0	0.00
crypto.rsa	35	414	16759	431	16	12	40	5	0	1.21
crypto.signverify	35	414	16747	430	16	12	40	11	0	2.66
derby	672	12361	373328	1418	942	18	30	886	57	7.63
mpegaudio	60	634	78498	450	16	11	124	65	0	10.25
scimark.fft.large	39	451	18253	424	16	12	40	1	0	0.22
scimark.lu.large	38	448	18220	424	16	12	41	1	0	0.22
scimark.monte_carlo	38	439	17657	425	16	12	40	1	0	0.23
scimark.sor.large	39	444	17836	425	16	11	40	1	0	0.23
scimark.sparse.large	38	440	17832	424	16	12	41	1	0	0.23
serial	56	517	18465	443	16	9	36	26	3	5.61
sunflow	130	1552	71403	350	17	12	46	165	13	11.47
xml.transform	109	1115	42867	533	16	10	38	85	3	7.89
xml.validation	37	428	16919	438	16	12	40	9	0	2.10

Table 6.5: Code profile for the SPECjvm2008 benchmarks as returned by JINE

provides the virtual throughputs, as they would be without any time scaling factor. Comparing the un-modified virtual throughput to the real one, we can extract quantitative and qualitative results on the accuracy and the overhead of VEX. Each benchmark in SPECjvm2008 is run with a 'warmup' (set to 24s in each experiment here) and is run for a fixed time (set to 96s in each experiment). To identify the overhead of VEX, the total execution time (including loading, instrumentation, warmup and main run) was measured externally through bash scripts.

The results for a single thread execution of each benchmark are shown in Table 6.6. The single-thread case might seem uninteresting from the point of view of a scheduler, but VEX still needs to synchronise more than one thread; specifically the threads that are spawned to run

Benchmark Threads: 1		Real time		VEX full profiling simulation				VEX adaptive profiling simulation				
Benchmark	**Meth**	**o/m**	**Time**	**o/m**	**Time**	**Err%**	**O/h**	**Meth**	**o/m**	**Time**	**Err%**	**O/h**
comp.compiler	4.8M	68.05	138.65	77.22	172.08	13.48	1.24	0.51M	77.36	173.66	13.68	1.25
comp.sunflow	11.65M	24.97	132.02	26.96	168.67	7.97	1.28	0.36M	27.11	168.63	8.57	1.28
compress	9.33B	35.18	125.62	10.22	5419.82	-70.95	43.14	0.39M	36.54	145.4	3.87	1.16
crypto.aes	0.02M	11.12	127.04	12.16	142.74	9.35	1.12	0.02M	12.15	146.19	9.26	1.15
crypto.rsa	0.36M	58.31	124.21	63.23	140.63	8.44	1.13	0.31M	63.5	141.93	8.9	1.14
crypto.signver	0.74M	61.38	124.26	66.97	139.52	9.11	1.12	0.41M	66.86	142.39	8.93	1.15
derby	5.46B	38.92	208.19	26.6	3696.11	-31.65	17.75	0.95M	38.77	325.49	-0.39	1.56
mpegaudio	1.8B	18.13	127.37	16.26	1175.23	-10.31	9.23	0.95M	19.98	129.13	10.2	1.01
sci.fft.large	58.7M	8.08	131.31	6.79	202.79	-15.92	1.54	0.02M	8.1	132.93	-3.34	1.02
sci.fft.small	0.51B	59.92	125.1	61.02	415.85	1.84	3.32	0.51M	64.1	166.98	6.98	1.33
sci.lu.large	29.4M	3.01	141.45	2.83	194.68	-5.85	1.37	0.01M	3.19	163.7	8.87	1.1
sci.lu.small	0.57B	68.1	124.98	70.21	439.45	3.1	3.52	0.52M	74.77	143.34	9.79	1.15
sci.monte_carlo	1.07B	32.09	124.86	7.41	3059.7	-76.91	24.51	0.17M	35.6	142.68	10.94	1.14
sci.sor.large	62.9M	9.39	128.98	7.21	205.31	-23.22	1.59	0.02M	9.89	131.57	5.32	1.02
sci.sor.small	82.05M	37.07	124.31	40.11	182.71	8.2	1.47	0.18M	40.49	140.35	9.23	1.13
sci.sparse.large	63.0M	9.03	124.51	7.16	207.67	-20.75	1.66	0.02M	9.72	134.16	5.19	1.08
sci.sparse.small	88.85M	36.19	126.3	36.58	186.26	1.08	1.47	0.17M	37.25	142.17	2.93	1.13
serial	15.51M	25.24	126.1	26.9	152.67	6.58	1.21	0.36M	27.46	143.24	8.8	1.14
sunflow	1.12B	9.31	127.7	0.48	2690.91	-94.84	21.07	13.04M	10.11	355.08	8.59	2.78
xml.transform	98.43M	49.35	149.64	55.53	243.22	12.52	1.63	10.81M	55.72	197.94	12.91	1.32
xml.validation	2.48M	68.75	124.43	74.6	146.62	8.51	1.18	0.41M	73.69	151.22	7.19	1.22
Geom. mean						11.55	2.81				6.58	1.22
Arithm. mean						20.98	6.74				7.8	1.25

Table 6.6: SPECjvm2008 benchmark results with a single workload-driving thread and no time scaling

each benchmark iteration (one per iteration), timer threads, and any additional background worker threads that the benchmarks (like *sunflow* or *derby*) might need.

The "Meth" column of Table 6.6 shows the number of method calls invoked during the program execution. The tables shows two sets of results: one in which all methods are instrumented ("full profiling") and the other in which the adaptive profiling has been used to remove instruments from short-running methods. The full-profiling results show a prediction error with a geometric mean of 11.55% and overhead of 2.81. As might be expected, the magnitude of the prediction error is found to be larger in benchmarks with a high turnover of short methods, which is the case with the *compress*, *derby*, *scimark.monte_carlo* and *sunflow* benchmarks. To enable the adaptive profiling scheme, we define, as with the other benchmarks of our evaluation study, a short running method to be one that executes more than a specified number times (here 10,000) within the (fixed) benchmark running time (here 96s). The number of methods profiled under the adaptive regime is also shown in the table (Second "Meth" column). As shown in the table, adaptive profiling significantly reduces both the prediction error (11.55% $\rightarrow$ 6.58%) and execution time overhead (2.81 $\rightarrow$ 1.22). On average, less than 1% of the methods were removed by the adaptive profiling, but these accounted for around 99% of the method calls made. It is

important to appreciate that, if the framework is to be used solely for performing time-scaling experiments, rather than for more conventional profiling purposes, then only the methods that need to be scaled will have to be instrumented.

The reader might note that in almost all cases the VEX results are *overpredicting* the throughput. There are three factors that contribute to this effect. First, the use of CPU-timers does not account for any background load. Secondly, system calls that take place internally within the native JVM library code (i.e. not those initiated at the application level) are not measured by CPU-time, as they happen in kernel space. Finally, the internal JVM threads (excluding GC threads which are profiled in real time) are neither profiled nor controlled by the virtual time framework. The first of these is the most significant, but we make no attempt to factor in any background load.

6.4.2 Multithreaded

Effect of VEX parameters

In this section we conduct a series of experiments to identify how the various VEX parameters affect the prediction accuracy and overheads of our methods for the SPECjvm2008 benchmarks. The benchmarks are executed with four threads on a uniprocessor simulation host (Host-3 from Appendix A with its second core disabled) and the adaptive profiling scheme with a threshold of 10,000 method invocations is by default turned on. We also use the default value of 100ms for the scheduler timeslice, i.e. the duration that the VEX scheduler waits in *real* time until it signals the currently running thread to find how much it has progressed in *virtual* time.

Using the default scheduler timeslice and profile aggregation at the method-level, we acquire the results under the corresponding column of Table 6.7. The magnitude of the prediction error is typically low with a geometric mean of 8.12% and an arithmetic mean of 12.87%. The highest errors and overheads are found in the two *comp.** benchmarks (and also the *xml.** benchmarks), due to an increased number of native waiting thread states. Recall that native waiting threads are only identified as such when their timeslice expires (see Section 5.2.2 for

Threads: 4	Real time		100ms timeslice on 1 core				10ms timeslice on 1 core			
Benchmark	**ops/m**	**Time**	**ops/m**	**Time**	**Err.%**	**O/h**	**ops/m**	**Time**	**Err.%**	**O/h**
comp.compiler	72.24	141.48	100.52	370.88	39.15	2.62	79.69	183.75	10.32	1.30
comp.sunflow	27.20	146.17	37.90	446.65	39.34	3.06	29.30	196.11	7.72	1.34
compress	41.07	138.55	42.97	172.24	4.63	1.24	42.83	158.09	4.29	1.14
crypto.aes	12.49	171.82	14.06	239.62	12.57	1.39	10.43	171.18	-16.47	1.00
crypto.rsa	67.20	126.84	68.24	135.26	1.55	1.07	68.43	136.52	1.83	1.08
crypto.signverify	69.34	128.39	70.08	140.95	1.07	1.10	73.78	156.68	6.40	1.22
derby	43.09	321.35	40.39	573.69	-6.27	1.79	41.65	450.61	-3.34	1.40
mpegaudio	22.29	141.10	26.97	247.10	21.00	1.75	26.19	209.52	17.50	1.48
sci.fft.large	7.74	199.65	8.86	196.88	14.47	0.99	8.69	197.14	12.24	0.99
sci.fft.small	67.35	128.38	74.34	283.14	10.38	2.21	72.63	157.74	7.83	1.23
sci.lu.large	2.89	392.25	3.21	397.22	11.07	1.01	3.14	403.17	8.56	1.03
sci.lu.small	74.44	129.78	84.92	531.05	14.08	4.09	82.76	186.94	11.17	1.44
sci.monte_carlo	43.16	134.95	45.42	147.91	5.24	1.10	46.14	143.83	6.90	1.07
sci.sor.large	9.37	168.11	9.90	223.19	5.66	1.33	9.84	188.98	4.99	1.12
sci.sor.small	43.27	137.07	45.62	146.69	5.43	1.07	45.33	148.18	4.77	1.08
sci.sparse.large	8.63	207.16	10.03	260.97	16.22	1.26	9.36	231.82	8.43	1.12
sci.sparse.small	35.43	137.77	35.30	198.73	-0.37	1.44	35.32	160.98	-0.31	1.17
serial	30.12	145.54	31.94	193.61	6.04	1.33	27.87	150.46	-7.49	1.03
sunflow	13.22	177.27	14.40	747.90	8.93	4.22	12.40	583.95	-6.18	3.29
xml.transform	52.57	158.52	62.78	547.00	19.42	3.45	62.77	294.82	19.41	1.86
xml.validation	75.33	134.50	96.00	351.53	27.44	2.61	80.02	158.79	6.23	1.18
Geo-mean					8.12	1.70			6.40	1.26
Mean					12.87	1.91			8.21	1.31

Table 6.7: SPECjvm2008 benchmark results for 4 threads with VEX scheduler timeslices of 100ms (default) and 10ms. Means of 3 runs are shown

details). At the stop-the-world stage of every garbage collection, the HotSpot JVM requires that all application threads are blocked at safepoints. From VEX's perspective they all have to become native waiting one-by-one (see Section 5.2.3), before the garbage collection may take place. The longer the timeslice, the longer it takes for a thread to be identified as native waiting, the longer the total safepoint synchronisation time and thus the higher the overhead of the simulation.

Reducing the scheduler timeslice to 10ms, we acquire the results of the last four columns of Table 6.7. This shows an improvement to the overheads of all benchmarks that suffer from the native waiting problem as we would expect. The prediction error is also reduced for the two *comp.** benchmarks from 39.15% and 39.34% to 10.32% and 7.72% respectively indicating that the higher overprediction is also attributed to native waiting threads. Overall, there is a significant improvement to the geometric mean of the simulation overhead (1.70 $\rightarrow$ 1.26) and to the prediction error (8.12% $\rightarrow$ 6.40%).

We next investigate whether there exists an "optimal" VEX scheduler timeslice that would lead to lower prediction errors and simulation overheads for the SPECjvm2008 benchmarks. The results for timeslices from 64 to 0.1 ms are shown in Figure 6.3. The optimal value for the

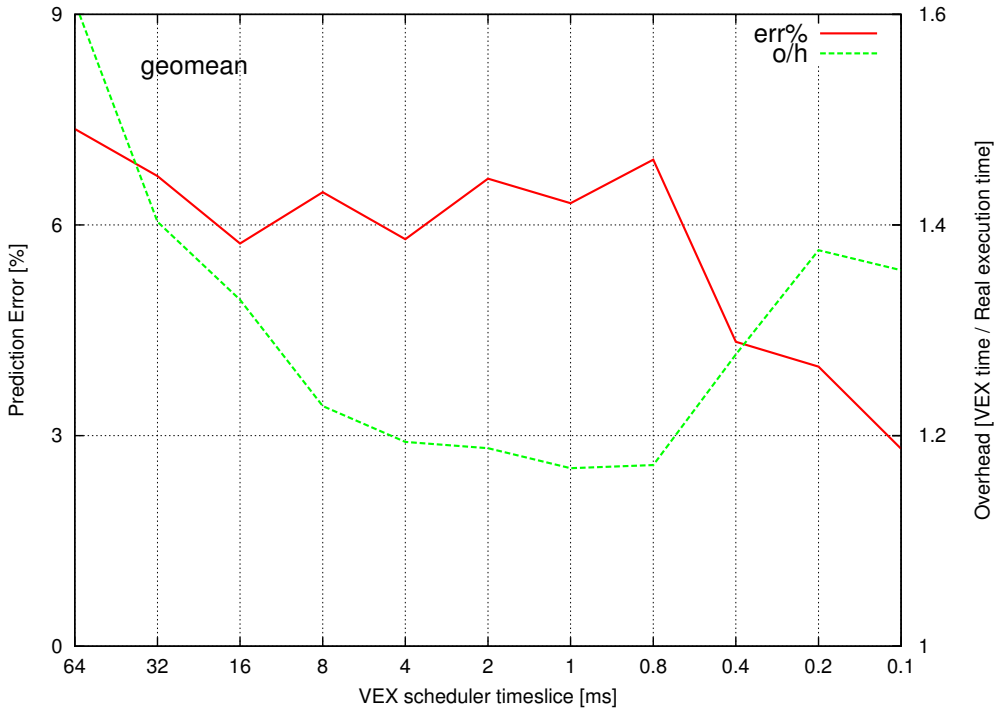


Figure 6.3: Effect of the VEX scheduler timeslice on the prediction error and simulation overhead of the SPECjvm2008 benchmarks with four benchmark threads. The geometric mean of all benchmark results (using absolute values for prediction errors) are shown.

simulation overhead is found to be 1.169 for a timeslice of 1ms. This is because, as we decrease the scheduler timeslice, we limit the simulation overhead, by reducing the times required to resume all threads suspended in VEX and thus allowing them to synchronise at safepoints faster (see Section 5.2.3). Indeed, using the "`-XX:+PrintSafepointStatistics`" JVM flag we verify this effect. On the other hand, reduced timeslices increase the rate at which the application threads are signalled by the VEX scheduler and become context switched - this increases the simulation overhead. Therefore a timeslice of 1ms comprises the "sweet-spot" between these two opposing effects.

Figure 6.3 also shows a decrease in the geometric mean of the prediction error as we reduce the scheduler timeslice beyond 0.8ms, with an optimal value of 2.8% error at 0.1ms. Prediction errors for timeslices larger than 0.8ms range between 6 and 7%. Further analysis indicates that benchmarks whose threads are considered "Native Waiting" at a higher rate (e.g. *compiler.compiler*, *compiler.sunflow*, *sunflow*) for timeslices greater than 0.8ms present the largest accuracy improvement as the timeslice decreases (see Figure 6.4). Recall that using the default "counter-based" criteria, a thread is identified as "Native waiting" when its CPU-time progress

is less than p percent of its last assigned scheduler timeslice s (see Section 5.2.2). This means that for lower values of s, even a low progress of a thread in terms of *absolute* CPU-time is enough for the thread *not* to be considered as "Native waiting". This leads to threads synchronising at safepoints without being regarded as "Native waiting", despite the low absolute CPU-time progress they present at the synchronisation phase. As "Native waiting" threads account for their time only when they return under VEX's control, they constitute a source of inaccuracy; this is significantly contained with lower scheduler timeslices, but at the cost of simulation overhead.

In most experiments of this chapter we have decided to use the timeslice of 10ms, which is a middle way between the two optima: it is more accurate than the fastest 1ms timeslice and faster than the most accurate 0.1ms.

Threads: 4	**Real time**		**10ms timeslice stack-trace level**				**10ms timeslice I/O threshold=50μs**			
Benchmark	**ops/m**	**Time**	**ops/m**	**Time**	**Err.%**	**O/h**	**ops/m**	**Time**	**Err.%**	**O/h**
comp.compiler	72.24	141.48	80.50	183.63	11.43	1.30	81.35	180.95	12.61	1.28
comp.sunflow	27.2	146.17	29.97	196.17	10.18	1.34	29.95	197.55	10.11	1.35
compress	41.07	138.55	42.74	150.11	4.07	1.08	43.33	146.47	5.50	1.06
crypto.aes	12.49	171.82	10.36	240.16	-17.05	1.40	11.34	236.32	-9.25	1.38
crypto.rsa	67.2	126.84	69.33	135.34	3.17	1.07	69.71	136.72	3.74	1.08
crypto.signverify	69.34	128.39	74.75	165.07	7.80	1.29	75.92	163.06	9.49	1.27
derby	43.09	321.35	44.95	485.32	4.32	1.51	43.35	432.37	0.60	1.35
mpegaudio	22.29	141.1	24.24	206.88	8.75	1.47	23.46	198.76	5.25	1.41
sci.fft.large	7.74	199.65	8.05	214.43	4.01	1.07	8.12	206.52	4.91	1.03
sci.fft.small	67.35	128.38	72.25	155.84	7.28	1.21	71.80	157.02	6.61	1.22
sci.lu.large	2.89	392.25	3.09	396.00	6.92	1.01	3.20	394.27	10.73	1.01
sci.lu.small	74.44	129.78	82.90	184.81	11.36	1.42	82.83	167.89	11.27	1.29
sci.monte_carlo	43.16	134.95	45.13	144.31	4.56	1.07	45.23	153.86	4.80	1.14
sci.sor.large	9.37	168.11	9.82	203.93	4.80	1.21	9.84	198.03	5.02	1.18
sci.sor.small	43.27	137.07	45.34	144.97	4.78	1.06	45.34	144.81	4.78	1.06
sci.sparse.large	8.63	207.16	9.61	216.87	11.36	1.05	9.41	215.26	9.04	1.04
sci.sparse.small	35.43	137.77	35.31	157.54	-0.34	1.14	35.32	160.19	-0.31	1.16
serial	30.12	145.54	28.47	194.75	-5.48	1.34	28.68	192.44	-4.78	1.32
sunflow	13.22	177.27	11.20	630.39	-15.30	3.56	13.22	607.49	0.10	3.43
xml.transform	52.57	158.52	63.42	311.45	20.64	1.96	63.97	284.49	21.69	1.79
xml.validation	75.33	134.5	79.05	157.86	4.94	1.17	78.92	156.47	4.77	1.16
Geo-mean					6.29	1.31			4.47	1.28
Mean					8.03	1.37			6.92	1.33

Table 6.8: SPECjvm2008 benchmark results for 4 threads with stack-trace level profiling and an I/O threshold of 50μs (using the *max* prediction method). Means of 3 measurements are shown.

In Table 6.8 we investigate the effect of two other VEX parameters. The first specifies that method profiles are aggregated on a per stack-trace level and the second that an I/O threshold is used to distinguish "Cached" from "Standard" I/O operations (see Section 4.1.2). The observed means of the VTEs using per-method or per-stack-trace profile aggregation are approximately equal for both prediction error and overhead.

We also explore how the SPECjvm2008 results are affected by using an I/O-threshold of I_T (here $I_T = 50\mu s$) and replaying the maximum observed I/O duration of the last $B = 16$ measurements (*max* I/O prediction policy) of each invocation point; recall that previous experiments (Section 4.3) have shown this configuration to be effective. Indeed the last four columns of Table 6.8 show that these parameters improve the geometric mean of the prediction error to 4.47%. This is mostly attributed to the *derby* benchmark, which performs the most invocations of methods of the `java.io` package (see Table 6.5). Without the threshold, all of them are recognised as "Standard I/O" operations that yield the CPU to other application threads and thus reduce the total execution time (and increase the throughput result). Using the threshold I_T, we limit this effect, whilst reducing the simulation overhead, by removing any additional handling for I/O requests that are predicted to be below I_T.

Effect of JVM performance tuning parameters

Next we evaluate how changing the parameters of the JVM affects the reproduction of real time results in virtual time. To investigate whether effects on the JVM execution are reflected in the virtual time execution, we use two parameters that affect garbage collection and a JIT-related parameter that enables the use of aggressive compiler optimisations. The first parameter enables the HotSpot concurrent mark sweep garbage collector, which minimises the pause times of each major collection by tracing most live objects and sweeping unreachable ones concurrently with the execution of the application. Since our experiments run on a uniprocessor, both application and garbage collection threads contend for the single CPU resource; this is expected to lower the observed throughput. Indeed, Table 6.9 shows the concurrent collector lowers the throughput of operations across all benchmarks that exhibit high garbage collection activity in both the RTE and the VTE. In fact, cases where usage of this collector (with the default heap size) led to out-of-memory exceptions, were similarly identified in both executions, though in the case of *scimark.sor.large* it took 4.66 times longer for VEX to do so. The results of such executions have not been taken into account in the prediction error measurements and, since the excluded benchmarks were accurately simulated in the previous experiments, the geometric mean of the prediction error is seen to increase to 9.22%.

	ConcurrentMarkSweep						2GB available heap to limit collections					
Threads: 4	**Real time**		**VEX**				**Real time**		**VEX**			
Benchmark	**o/m**	**Time**	**o/m**	**Time**	**Err%**	**O/h**	**o/m**	**Time**	**o/m**	**Time**	**Err%**	**O/h**
comp.compiler	47.59	154.11	59.59	227.47	25.22	1.48	99.16	143.84	118.35	177.39	19.36	1.23
comp.sunflow	18.44	160.03	23.82	239.68	29.18	1.50	39.85	154.03	45.60	182.03	14.43	1.18
compress	41.04	138.63	42.78	150.60	4.24	1.09	40.18	142.72	42.71	149.93	6.30	1.05
crypto.aes	12.54	184.79	13.86	250.11	10.53	1.35	12.98	178.44	13.96	257.95	7.55	1.45
crypto.rsa	63.16	129.39	68.62	134.92	8.64	1.04	69.46	129.68	69.14	141.11	-0.46	1.09
crypto.signverify	69.58	127.99	75.41	158.63	8.38	1.24	71.06	129.24	74.07	157.81	4.24	1.22
derby	OOM	165.68	OOM	214.36	N/A	1.29	43.53	284.97	41.08	434.08	-5.63	1.52
mpegaudio	20.45	148.61	25.93	200.60	26.80	1.35	21.64	145.42	25.87	211.05	19.55	1.45
sci.fft.large	OOM	24.69	OOM	25.65	N/A	1.04	8.44	188.62	8.05	215.08	-4.62	1.14
sci.fft.small	66.60	129.57	73.03	163.27	9.65	1.26	68.52	129.32	71.60	153.10	4.50	1.18
sci.lu.large	OOM	23.27	OOM	24.78	N/A	1.06	2.96	365.17	3.14	384.33	6.08	1.05
sci.lu.small	74.50	129.14	83.19	185.12	11.66	1.43	79.40	127.27	82.78	185.09	4.26	1.45
sci.monte_carlo	43.19	138.88	45.45	143.61	5.23	1.03	39.78	130.12	46.40	141.10	16.64	1.08
sci.sor.large	OOM	34.32	OOM	159.93	N/A	4.66	9.37	174.87	9.83	198.50	4.91	1.14
sci.sor.small	43.11	137.22	45.34	144.97	5.17	1.06	42.84	138.31	45.34	144.70	5.84	1.05
sci.sparse.large	OOM	21.11	OOM	23.12	N/A	1.10	8.93	180.59	9.49	181.32	6.27	1.00
sci.sparse.small	34.33	138.67	35.35	201.38	2.97	1.45	37.10	133.70	35.24	159.12	-5.01	1.19
serial	OOM	8.51	OOM	10.39	N/A	1.22	30.33	145.06	29.07	160.37	-4.15	1.11
sunflow	13.15	178.11	13.36	581.17	1.60	3.26	14.00	168.40	13.16	603.17	-6.00	3.58
xml.transform	50.51	157.23	63.89	295.43	26.49	1.88	59.00	153.86	63.56	288.24	7.73	1.87
xml.validation	63.94	132.19	71.74	169.74	12.20	1.28	84.82	128.51	88.99	155.12	4.92	1.21
Geo-mean					9.22	1.40					6.00	1.28
Mean					12.53	1.53					7.54	1.35

Table 6.9: SPECjvm2008 benchmark results for 4 threads with the Concurrent-Mark-Sweep garbage collector or a 2GB large heap. Means of 3 measurements are shown.

The idea for the next experiment is to limit the effect of the garbage collector on our benchmarks by setting the minimum JVM-available heap to 2GB. This reduces the expected number of garbage collections, but may bear a negative impact on the locality of references. We expect both effects to be reflected in the VTE. The results of this experiment are shown in the last four columns of Table 6.9 and demonstrate a prediction error (6.00%) and overhead (1.28), very similar to the previous results.

In Table 6.11 we present the results, when the "aggressive optimisations" flag of the JVM is set. This enables a number of experimental performance optimisations, which, by examining the source of the HotSpot version we are using (1.6.0_23-b05), we find to be related to escape analysis, biased locking and string concatenation optimisation. We repeat the experiments for four benchmark threads using a 10ms VEX scheduler timeslice on a uniprocessor and find the geometric mean of the prediction errors to remain consistent around the 6% mark (5.59%) and the 1.20 overhead (1.19).

Based on the behaviour of the *compiler.compiler* benchmark that exhibits different performance results for each of the above three JVM parameters (99.16 → 69.19 → 47.59 ops/min for the

	-XX:+AggressiveOpts					
Threads: 4	**Real time**		**VEX**			
Benchmark	**ops/m**	**Time**	**ops/m**	**Time**	**Err.%**	**O/h**
comp.compiler	69.19	145.42	80.70	185.87	16.63	1.27
comp.sunflow	28.07	153.78	29.42	195.88	4.81	1.27
compress	41.78	136.83	41.08	153.97	-1.68	1.13
crypto.aes	12.31	167.08	13.59	181.05	10.40	1.08
crypto.rsa	69.22	129.79	67.77	136.11	-2.09	1.05
crypto.signverify	68.27	130.10	73.55	142.35	7.73	1.09
derby	42.53	330.12	40.93	457.71	-3.76	1.39
mpegaudio	22.34	140.32	25.87	205.59	15.80	1.47
sci.fft.large	7.69	201.14	8.82	191.78	14.69	0.95
sci.fft.small	67.13	128.82	71.55	139.72	6.58	1.08
sci.lu.large	2.91	376.01	3.19	382.22	9.62	1.02
sci.lu.small	74.59	128.24	83.50	139.36	11.95	1.09
sci.monte_carlo	44.11	135.12	45.42	146.01	2.97	1.08
sci.sor.large	8.61	175.61	9.85	214.36	14.40	1.22
sci.sor.small	43.28	137.02	44.87	140.88	3.67	1.03
sci.sparse.large	8.94	174.65	9.56	219.43	6.94	1.26
sci.sparse.small	36.06	137.63	35.28	157.09	-2.16	1.14
serial	29.28	141.78	31.00	150.23	5.87	1.06
sunflow	12.98	179.86	12.77	347.31	-1.62	1.93
xml.transform	53.36	158.30	64.10	236.50	20.13	1.49
xml.validation	89.55	127.49	90.21	147.46	0.74	1.16
Geo-mean					5.59	1.19
Mean					7.82	1.20

Table 6.10: SPECjvm2008 benchmark results for 4 threads with the *-XX:+AggressiveOpts* optimisation options enabled. Means of 3 measurements are shown

huge heap, aggressive optimisation and concurrent-mark sweep respectively) we find that VEX reflects these changes in the VTE (118.35 $\rightarrow$ 80.70 $\rightarrow$ 59.59 ops/min) within a 25% over-prediction error.

Effect of JVM

We now explore the effect of changing the JVM. We use the 64-bit J9 IBM JVM (build 2.4, JRE 1.6.0) with the default parameters on the uniprocessor Host-3. The VEX scheduler timeslice is set to 10ms and returns per-method profiles, with an adaptive profiling policy at $I = 10,000$ method invocations. The results for the IBM JVM are shown in Table 6.11 and generally present low prediction errors (with 80th percentile of the absolute error being 20.34%) that are bound at a rather higher overhead of 1.94. Although the geometric mean of the absolute prediction error remains reasonably low at 10.53%, we identify noticeable differences between the IBM and the HotSpot JVMs not only in terms of observed throughputs (like for benchmarks *derby*, $sci.monte-carlo$, *serial* and $xml.transform$), but also for prediction errors: in cases like the two *xml.** benchmarks or the *sunflow* benchmark, we have very high errors as a

result of a high number of native waiting threads. Specifically, most threads are waiting at a JVM-internal conditional variable invoked by the native methods `javaCheckAsyncMessages` or `jitAcquireVMAccess`. These waiting events are not notified to VEX, thus leading to a very low CPU utilisation and high simulation overheads (ranging from 3.66 to 5.08) for these benchmarks. This study once again demonstrates how the inner workings of the JVM affect the virtual time execution.

	IBM JVM					
Threads: 4	**Real time**		**VEX**			
Benchmark	**ops/m**	**Time**	**ops/m**	**Time**	**Err.%**	**O/h**
comp.compiler	58.62	143.05	49.12	224.26	-16.2	1.57
comp.sunflow	24.1	149.4	19.2	265.52	-20.34	1.78
compress	42.31	132.69	41.61	178.14	-1.66	1.34
crypto.aes	20.92	148.03	23.1	243.71	10.42	1.65
crypto.rsa	53.12	133.06	55.68	167.99	4.81	1.26
crypto.signverify	53.14	134.53	55.76	382.32	4.93	2.84
derby	12.47	358.22	8.04	510.53	-35.52	1.43
mpegaudio	21.27	158.57	23.27	521.61	9.38	3.29
sci.fft.large	7.88	190.36	7.91	214.39	0.43	1.13
sci.fft.small	66.08	130.44	69.71	553.94	5.49	4.25
sci.lu.large	2.86	375.84	3.18	371.79	11.18	0.99
sci.lu.small	73.33	128.88	66.82	550.03	-8.88	4.27
sci.monte_carlo	27.36	142.71	22.72	163.1	-16.97	1.14
sci.sor.large	8.78	184.25	10.28	208.18	17.08	1.13
sci.sor.small	42.63	136.87	45.36	147.14	6.4	1.08
sci.sparse.large	8.6	176.29	9.6	220.05	11.59	1.25
sci.sparse.small	31.82	137.14	35.37	300.62	11.16	2.19
serial	16.02	165.46	18.88	288.75	17.85	1.75
sunflow	11.66	163.93	6.15	833.06	-47.21	5.08
xml.transform	29.91	175.52	17.53	716.29	-41.38	4.08
xml.validation	62.35	130.52	41.69	477.46	-33.14	3.66
Geo-mean					10.53	1.94
Mean					15.81	2.24

Table 6.11: SPECjvm2008 benchmark results for 4 threads executed by the IBM JVM. Means of 3 measurements are shown

Time scaling

The last four columns of Table 6.12 investigate the time scaling capabilities of VEX. For each benchmark we accelerate the `executeIteration()` method, which has the effect of accelerating the entire code by a factor of two [1]. Accordingly, the expected throughput should be approximately double the original throughput for four threads. The results in the "Pred/RT" column show that the scaling of almost all benchmarks resulted in approximate doubling of the throughput. Due to the fact that we are measuring throughput over a predefined period of

[1]Since in the *sunflow* benchmark the `executeIteration()` method merely spawns internal threads to execute the load, we scale instead the time of the `org/sunflow/core/renderer/BucketRenderer.renderBucket` method, which is invoked by these threads and accounts for their entire execution time.

Threads: 4	Real time		VEX adaptive profiling 10ms				VEX x2 accelerated simulation			
Benchmark	**ops/m**	**Time**	**ops/m**	**Time**	**Err.%**	**O/h**	**ops/m**	**Time**	**Pred./RT**	**O/h**
comp.compiler	72.24	141.48	79.69	183.75	10.32	1.30	121.57	265.92	1.68	1.88
comp.sunflow	27.20	146.17	29.30	196.11	7.72	1.34	42.40	272.73	1.56	1.87
compress	41.07	138.55	42.83	158.09	4.29	1.14	85.18	293.48	2.07	2.12
crypto.aes	12.49	171.82	10.43	171.18	-16.47	1.00	26.53	437.63	2.12	2.55
crypto.rsa	67.20	126.84	68.43	136.52	1.83	1.08	132.03	249.26	1.96	1.97
crypto.signverify	69.34	128.39	73.78	156.68	6.40	1.22	143.25	294.90	2.07	2.30
derby	43.09	321.35	41.65	450.61	-3.34	1.40	89.27	615.70	2.07	1.92
mpegaudio	22.29	141.10	26.19	209.52	17.50	1.48	48.93	342.07	2.20	2.42
sci.fft.large	7.74	199.65	8.69	197.14	12.24	0.99	16.02	337.88	2.07	1.69
sci.fft.small	67.35	128.38	72.63	157.74	7.83	1.23	146.03	335.39	2.17	2.61
sci.lu.large	2.89	392.25	3.14	403.17	8.56	1.03	6.60	529.34	2.28	1.35
sci.lu.small	74.44	129.78	82.76	186.94	11.17	1.44	166.82	445.65	2.24	3.43
sci.monte_carlo	43.16	134.95	46.14	143.83	6.90	1.07	90.63	271.93	2.10	2.02
sci.sor.large	9.37	168.11	9.84	188.98	4.99	1.12	19.78	320.51	2.11	1.91
sci.sor.small	43.27	137.07	45.33	148.18	4.77	1.08	90.79	273.70	2.10	2.00
sci.sparse.large	8.63	207.16	9.36	231.82	8.43	1.12	18.81	387.31	2.18	1.87
sci.sparse.small	35.43	137.77	35.32	160.98	-0.31	1.17	70.99	320.98	2.00	2.33
serial	30.12	145.54	27.87	150.46	-7.49	1.03	62.75	305.69	2.08	2.10
sunflow	13.22	177.27	12.40	583.95	-6.18	3.29	27.05	1236.83	2.00	6.98
xml.transform	52.57	158.52	62.77	294.82	19.41	1.86	116.75	435.92	2.22	2.75
xml.validation	75.33	134.50	80.02	158.79	6.23	1.18	151.07	294.03	2.01	2.19
Geo-mean					6.40	1.26			2.05	2.25
Mean					8.21	1.31			2.06	2.39

Table 6.12: SPECjvm2008 results for 4 threads with time scaling factors of 1.0 and 2.0. Means of 3 runs are shown

time, the simulation duration has doubled as well, as time flows slower in the virtual timeline. The predicted throughput of both *compiler* benchmarks is less than double. We attribute this to the significant time of garbage collection in their execution. In the current framework GC is not accelerated in virtual time.

Thread scaling

The final set of experiments investigates how VEX predictions and simulation overheads scale with an increasing number of threads. Table 6.13 shows the results for 1 to 16 threads in three different modes of VEX simulation:

- Adaptive Profiling (AP): the typical simulation with an adaptive profiling threshold of 10,000 method invocations and a 10ms scheduler timeslice (to speed up the experiment execution)
- No Profiling (NP): VEX simulation without any profiling instruments. Only thread-state monitoring instruments are added. As this approach significantly limits the points where native waiting threads can be recognised as such, we need to set the "poll_nw"

flag of VEX (see Section 5.2.2) to avoid threads executing the entire benchmark as native waiting. Accordingly, the VEX scheduler signals all threads that are considered native waiting before selecting the next thread to resume. This mode investigates VEX without the overhead and "observer effects" caused by the addition of method entry and exit instruments.

- No Scheduling (NS): Benchmark execution with VEX instruments, but without a VEX scheduler. Adaptive profiling with a 10,000 method invocation threshold is still enforced. Threads execute freely according to the schedule defined by the OS, but times are accounted in the same way as they would be in VEX (i.e CPU-time for non-I/O threads and wallclock time for I/O threads). This mode investigates VEX without the overhead and "observer effects" caused by the scheduling of threads by the VEX scheduler. It only works for a uniprocessor execution, where all thread progress in CPU-time is mapped onto the same virtual timeline.

The results are encouraging in that most VTEs present low prediction errors at low overheads. In particular the AP mode that enforces normal VEX simulation with profiling presents a prediction error of a 5.23% median (13.46% 95th percentile), a geometric mean of 5.59% and an overhead of 1.26 across the various selections for benchmark threads for this configuration.

The NP mode presents slightly higher prediction errors (6.58%) and overheads (geometric mean of geometric means equal to 1.55 instead of 1.26), despite the lack of profiling instruments. The main issue is the increased number of signals due to the "poll_nw" parameter: in benchmarks with significant garbage collections (and thus native waiting states) and more than 8 threads, we find overheads as high as 21.04 and 28.12 times the duration of the RTE. It is evident that the use of "poll_nw" does not scale well: the geometric mean of the simulation overhead increases from 1.14 for 1 thread to 2.53 for 16 threads (1.16 to 4.57 for arithmetic mean). However, recall that "poll_nw" signals are only used as a way of dealing with native waiting threads: properly identifying the latter (see Section 9.3.1) would allow a faster simulation without profiling instruments for cases where the performance testing interest lies only in the total application response time or throughput and not in the per-method times.

The NS mode bypasses the VEX scheduler and is used to identify the effect of VEX scheduling on prediction accuracy, whilst retaining the same number of profiling instruments as the normal adaptive profiling VEX simulation (AP). The prediction error increases with the number of threads (from 1.74% for 1 thread $\rightarrow$ 6.07% for 16 threads), due to the higher context switching overheads, that are not accounted for in the VTE. In any case, the overhead remains quite low (with a geometric mean of 1.06), showing that the mere existence of profiling instruments with our adaptive profiling policy enforced does not significantly affect simulation time. This is consistent with our observations on the single-threaded JGF benchmarks.

The conclusions from the SPECjvm2008 benchmarks are the following:

- The prediction error and overhead of VEX and JINE remain typically within 10% and 1.3 even for benchmarks of larger size and for various VEX, JVM parameters, time and thread scaling.
- Native waiting threads (typically due to GC and safepoint synchronisation) affect both the overhead and the prediction accuracy of the benchmarks.
- The VLF avoidance scheme is effective: no premature leaps have been observed for this benchmark suite for the HotSpot JVM.
- The results on the IBM JVM show reasonably low but slightly higher prediction errors and overheads than the HotSpot underlining once again how the inner workings of a JVM affect the virtual time execution.
- JIT effects are still present in larger benchmarks, in some cases reducing the accuracy of the VEX execution.
- Adaptive profiling on its own has a limited effect on the simulation overhead, as demonstrated by the thread scaling results in the NS mode.
- The fact that the NS results are slightly better than those of NP, suggests that the VEX scheduling has a greater impact on the prediction accuracy than profiling instruments.

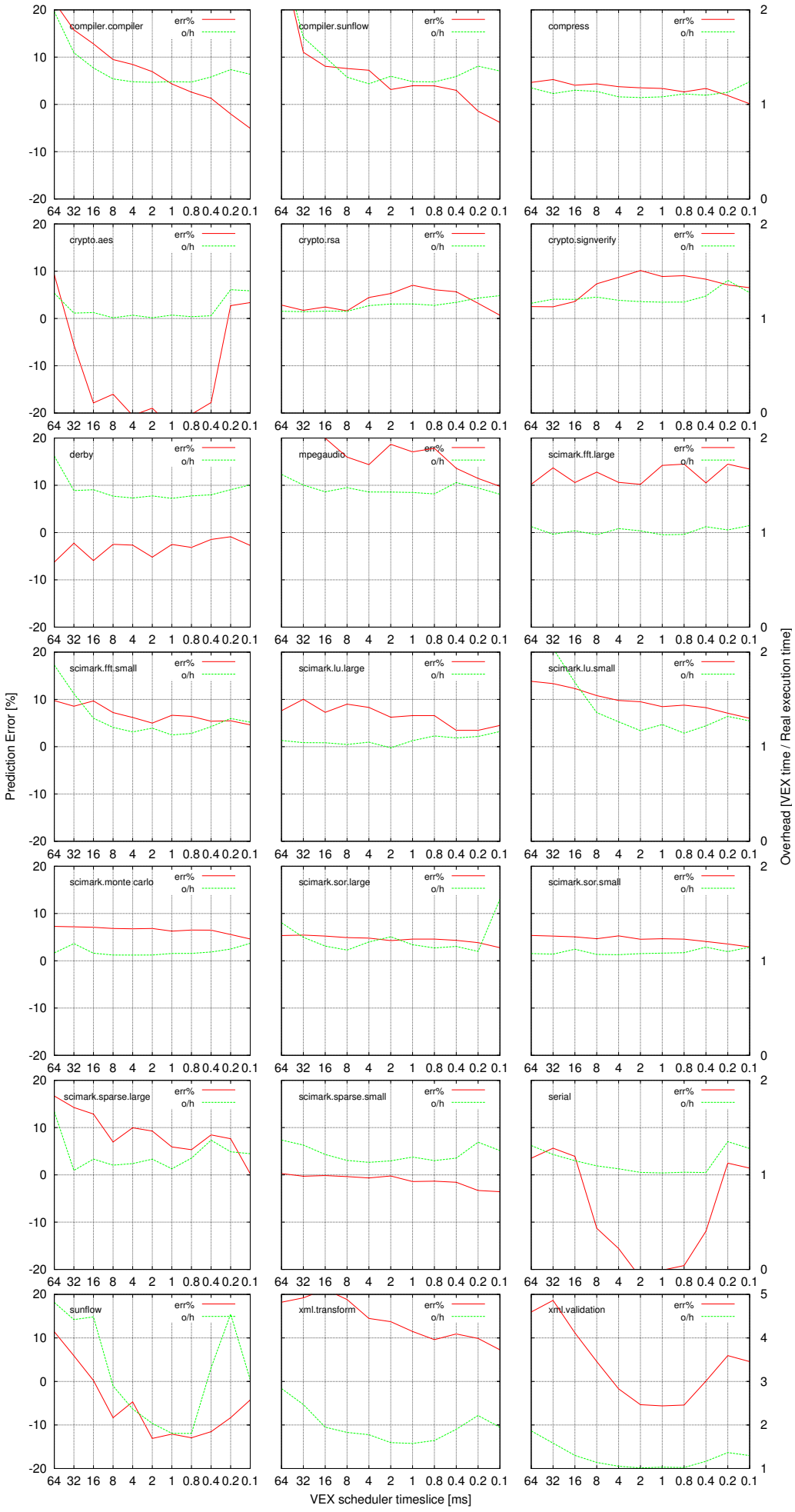


Figure 6.4: VEX scheduler timeslice effect on SPECjvm2008 benchmarks with four threads

		Error % for 1 to 16 threads						Overhead for 1 to 16 threads					
Benchmark	**Mode**	**1T**	**2T**	**4T**	**8T**	**16T**	**G-M**	**1T**	**2T**	**4T**	**8T**	**16T**	**G-M**
comp.compiler	AP	-11.31	0.14	10.32	25.43	59.95	7.61	1.14	1.16	1.30	1.19	1.25	1.21
	NP	13.23	-0.22	-3.74	8.24	37.34	5.07	1.32	1.23	1.32	1.56	2.47	1.53
	NS	-5.87	-4.07	-3.10	-9.45	16.32	6.48	1.01	1.02	1.06	0.97	1.14	1.04
comp.sunflow	AP	-5.91	-10.39	7.72	11.38	5.65	7.88	1.11	1.09	1.34	1.21	1.09	1.17
	NP	19.00	-3.21	-1.58	-3.05	-5.98	4.45	1.30	1.25	1.53	1.61	2.74	1.61
	NS	-13.53	-5.11	1.10	14.27	5.37	5.67	1.00	1.01	1.03	1.03	1.10	1.03
compress	AP	2.03	3.70	4.29	3.64	1.51	2.82	1.09	1.09	1.14	1.12	1.07	1.10
	NP	5.63	6.38	8.35	8.43	5.36	6.71	1.07	1.06	1.05	1.22	1.35	1.14
	NS	-2.75	-0.24	-2.61	8.94	-4.24	2.31	1.03	1.01	0.99	1.06	0.91	1.00
crypto.aes	AP	8.18	-5.14	-16.47	-17.75	-7.20	9.76	1.09	1.14	1.00	1.85	2.72	1.44
	NP	8.33	8.70	6.97	5.21	2.48	5.79	1.17	1.08	1.35	1.77	2.75	1.53
	NS	3.14	-1.11	16.17	18.65	39.12	8.37	0.98	0.99	1.08	0.96	1.00	1.00
crypto.rsa	AP	4.90	2.97	1.83	8.04	6.90	4.30	1.08	1.09	1.08	1.20	1.07	1.10
	NP	6.82	4.85	-4.81	-2.75	-5.93	4.82	1.08	1.09	1.03	0.98	0.97	1.03
	NS	-0.09	-1.72	0.45	8.07	3.81	1.16	1.00	1.01	1.03	1.04	1.00	1.02
crypto.signverify	AP	8.36	9.29	6.40	5.20	8.02	7.30	1.08	1.15	1.22	1.12	1.09	1.13
	NP	9.15	8.63	-2.32	-13.65	-4.10	6.34	1.09	1.13	1.13	0.99	1.24	1.11
	NS	2.84	-1.25	0.95	0.56	5.49	1.60	1.01	1.03	1.02	1.04	0.97	1.01
derby	AP	-8.25	-6.02	-3.34	-8.87	-2.67	5.24	1.35	1.45	1.40	1.46	1.33	1.40
	NP	-7.84	-9.74	-0.05	-7.04	-21.98	3.52	1.09	1.19	1.12	1.26	1.97	1.29
	NS	-18.54	-27.04	-18.77	-13.64	10.47	16.81	1.09	1.31	1.09	1.34	1.17	1.20
mpegaudio	AP	4.76	20.71	17.50	5.00	10.38	9.78	1.26	1.40	1.48	1.56	1.38	1.41
	NP	4.90	4.75	6.15	12.75	4.65	6.10	1.12	1.13	1.27	1.64	3.10	1.52
	NS	-6.72	-3.93	0.94	19.61	23.93	6.51	0.98	1.05	1.17	1.22	1.34	1.15
sci.fft.large	AP	-3.34	13.48	12.24	-4.26	OOM	8.20	1.02	1.09	0.99	1.12	1.07	1.06
	NP	5.73	13.48	12.92	-4.97	OOM	9.99	1.06	1.07	0.98	1.14	1.17	1.08
	NS	-3.10	15.73	12.53	8.05	OOM	8.49	0.97	1.09	1.00	1.07	1.11	1.05
sci.fft.small	AP	3.73	6.79	7.83	8.47	10.68	7.09	1.08	1.13	1.23	1.26	1.63	1.25
	NP	6.14	6.47	-7.66	9.02	9.71	7.68	1.06	1.11	1.06	1.70	4.64	1.58
	NS	-3.64	-1.13	1.14	6.84	6.19	2.88	1.01	1.00	1.02	0.98	1.02	1.01
sci.lu.large	AP	8.87	18.73	8.56	6.73	OOM	11.25	1.10	1.06	1.03	1.02	1.45	1.12
	NP	6.48	19.10	11.42	7.07	OOM	11.22	1.03	1.09	1.26	1.24	21.04	2.06
	NS	6.48	16.85	1.04	15.49	OOM	4.84	0.96	1.00	0.96	1.00	0.87	0.96
sci.lu.small	AP	3.25	9.10	11.17	14.82	15.16	9.42	1.13	1.22	1.44	1.32	2.58	1.46
	NP	6.30	10.09	11.71	11.99	13.94	10.45	1.07	1.25	1.51	3.03	8.07	2.18
	NS	-0.39	-7.62	0.75	3.48	12.40	2.49	1.00	1.01	0.98	1.00	0.97	0.99
sci.monte-carlo	AP	5.28	-6.64	6.90	4.83	3.88	5.39	1.08	1.08	1.07	1.10	1.14	1.09
	NP	5.36	5.45	4.94	4.51	3.93	4.80	1.07	1.06	1.17	1.14	1.22	1.13
	NS	0.16	-9.76	4.26	4.05	5.72	2.73	1.01	1.02	1.02	1.07	1.10	1.04
sci.sor.large	AP	5.32	5.12	4.99	4.90	4.60	4.98	1.02	1.02	1.12	1.07	1.07	1.06
	NP	5.32	5.33	5.12	5.22	8.77	5.82	1.02	1.01	1.30	1.74	2.16	1.38
	NS	-0.11	6.08	6.30	6.39	6.84	2.82	0.96	1.01	1.01	1.11	0.99	1.01
sci.sor.small	AP	5.24	13.60	4.77	4.71	4.05	5.79	1.06	1.07	1.08	1.10	1.05	1.07
	NP	5.29	13.50	4.92	4.88	4.35	5.95	1.06	1.06	1.05	9.79	1.08	1.66
	NS	-0.05	-0.02	-3.95	4.00	4.60	0.61	0.99	0.99	1.01	1.01	0.95	0.99
sci.sparse.large	AP	5.19	6.04	8.43	6.46	4.44	5.97	1.08	1.04	1.12	1.02	0.99	1.05
	NP	3.57	9.44	14.02	4.81	41.23	9.87	1.09	1.15	0.86	2.13	5.52	1.66
	NS	-2.71	-15.03	1.39	14.44	25.16	7.29	1.02	1.12	1.00	1.17	1.09	1.08
sci.sparse.small	AP	-7.12	-1.18	-0.31	-0.28	-1.21	0.98	1.07	1.10	1.17	1.23	1.47	1.20
	NP	4.69	9.57	10.47	10.67	8.68	8.47	1.11	1.13	1.20	1.60	3.31	1.51
	NS	-12.65	-9.01	-6.52	-0.74	-0.51	3.08	0.99	1.01	1.06	1.10	1.13	1.06
serial	AP	5.99	3.99	-7.49	0.10	-2.09	2.06	1.17	1.26	1.03	1.28	1.29	1.20
	NP	6.42	8.55	9.23	2.75	-3.42	5.44	1.14	1.19	1.39	1.21	1.09	1.20
	NS	-0.70	0.64	-0.40	-1.70	-5.06	1.09	0.99	1.03	0.94	0.93	0.95	0.97
sunflow	AP	5.59	8.89	-6.18	5.65	6.56	6.48	1.99	2.34	3.29	5.50	3.68	3.15
	NP	26.53	14.16	10.36	8.44	9.24	12.48	2.06	4.36	9.59	21.59	28.12	8.78
	NS	-31.19	-31.16	-24.36	0.23	-39.97	11.70	1.07	1.18	1.47	1.88	4.52	1.74
xml.transform	AP	10.24	22.56	19.41	13.12	11.60	14.68	1.55	1.76	1.86	1.51	1.39	1.60
	NP	8.79	21.07	10.82	-2.18	-5.89	7.63	1.08	1.32	9.91	1.01	0.98	1.69
	NS	0.60	1.31	-0.13	5.36	9.24	1.39	1.25	1.24	1.21	1.24	1.20	1.23
xml.validation	AP	10.63	2.86	6.23	-0.52	-1.53	2.73	1.19	1.17	1.18	1.05	0.95	1.10
	NP	8.82	1.66	4.81	-9.75	-11.06	5.97	1.17	1.31	1.35	1.07	1.06	1.19
	NS	0.54	-2.22	2.88	-0.05	-0.20	0.51	1.01	1.00	0.97	1.02	1.01	1.00
Geo-mean	AP	5.85	5.82	6.40	4.64	5.41	5.59	1.16	1.21	1.26	1.31	1.36	1.26
	NP	7.30	6.65	5.25	6.18	7.82	6.58	1.14	1.22	1.46	1.76	2.53	1.55
	NS	1.74	3.17	2.31	4.15	6.07	3.17	1.01	1.05	1.05	1.09	1.12	1.06
Mean	AP	6.36	8.44	8.21	7.63	8.85	7.90	1.18	1.24	1.31	1.44	1.46	1.33
	NP	8.30	8.78	7.25	7.02	11.04	8.48	1.16	1.30	2.02	2.83	4.57	2.38
	NS	5.51	7.67	5.23	7.81	10.26	7.30	1.01	1.05	1.05	1.11	1.22	1.09

Table 6.13: SPECjvm2008 benchmark results for 1 - 16 threads executing in various modes: adaptive profiling (AP), no profiling (NP) and no scheduling (NS)

Chapter 7

Multicore VEX

In this chapter we extend VEX to simulate multicore architectures, i.e. to provide performance predictions for a set of virtual performance specifications for a multicore target with m *virtual cores*. Each virtual core allows the virtual time execution of one thread, leading to up to m threads that can run in parallel within the same virtual timeslice. Each virtual core has its own virtual timeline, which is updated by the next runnable thread that is scheduled on that core. VEX allows only one thread per virtual core to be runnable and it is up to the OS scheduler to decide when and how to schedule such runnable threads on the system. By using CPU-time counters to measure thread progress, VEX results do not include the times that these runnable threads were waiting to be scheduled by the OS. This is thus a generalisation of the single CPU case presented in Section 3.1. For the most part of this chapter we assume that m is equal to the number of physical cores on the simulation host n. Accordingly, the number of threads that VEX will allow to be runnable is limited by the number of physical cores on the simulation host. In Section 7.3.3 we also explore the idea of using VEX to simulate more cores than the ones physically available, i.e. the case where $m > n$.

The main challenge of using VEX for simulating a multicore architecture lies in the fact that real time and virtual time progress simultaneously, but potentially at a different pace. For example, virtual time may progress faster or slower than real time when a time scaling factor is used, or the virtual time of a thread can be updated to a value arbitrarily later in time

when a virtual leap forward occurs. As a result a thread running on one of the m virtual cores may be executing code that is ahead in virtual time in parallel with code that is in the virtual past. This is tantamount to the *causality errors* presented in parallel discrete-event simulations (PDES) [47]. Such errors are not possible in the uniprocessor case, because the next runnable thread is always resumed at the virtual timestamp that the previous one released the CPU.

In order to keep the threads that are simulated in virtual time on m cores synchronised, we employ two different synchronisation techniques. The *Lax* synchronisation model assumes that threads execute on the m cores at the same pace, unless one of them uses time-scaling, leaps forward in virtual time or simulates a performance model (details on simulating performance models are provided in Chapter 8). Only then does VEX take synchronisation actions. This approach is expected to limit synchronisation needs between cores, but at the loss of some simulation inaccuracy. We also introduce the *Strict Parallel Execution* (SPEX) synchronisation model, which synchronises threads on every VEX scheduler timeslice. This imposes a stricter control on the progress of the simulation and is thus expected to be more accurate than the Lax model, but at a higher simulation overhead.

We note here that the problem of synchronising parallel processes and dealing with causality errors in computer simulation is not by any means new. Fujimoto [47] presents a survey of both conservative [28] and optimistic [70] synchronisation techniques for PDES. Research from execution-driven simulation has also used lax [29, 95], time-quanta [113, 98, 95], or barrier [152, 95] synchronisation approaches to control the progress of the parallel processes participating in a simulation. However, such methods do not assume any relation between real and simulation time progress, nor do they measure and depend on real time events (like I/O operations) on the simulation host.

Methods from the field of network emulation that focus on synchronising events in simulation time (network protocols) with ones happening in real time (application prototypes) are more closely related to the objectives of multicore virtual time execution. Such methods synchronise network protocols and application prototypes either using a single time-dilation factor across all virtual nodes participating in the simulation according to the availability of physical resources

[56], or adapting this factor according to the load in an epoch-based way [53]. Epochs are also used by Bergstrom et al to synchronise in relativistic time (i.e. scaled real time) and allow time warps if no activity is observed, while Weingaertner et al [142, 143] make use of a scheduler timeslice that controls the execution progress of the simulator and the virtual hosts. Most of these methods are similar to the ones proposed in this chapter, because they are categorised under the conservative time window approach [82], where the difference in the simulation time between two concurrently processed events is bound by a known constant. However, our focus here is on fitting such techniques to the particular characteristics of the multicore virtual time execution, i.e. the generally tight coupling between real and virtual time, the control of the simulation at the thread level, the scheduling of threads in virtual timeslices, the existence of profiling and state changing instruments, the dependence on real time events (like I/O operations) and the high level at which VEX is functioning.

In this chapter we present the Lax and SPEX synchronisation methods and discuss how the multicore virtual time execution affects other aspects of our framework, like I/O operations (see Chapter 4) or JINE implementation issues (see Section 5). The two synchronisation policies are applied on the multi-threaded benchmarks of the Java Grande Forum [68] and the SPECjvm2008 ones [128] that were also used in Chapter 6. In both cases, the simulation and the target host have the same number of cores $m = n = 2$. The main objective of these experiments is to evaluate the synchronisation policies in terms of simulation overhead and prediction accuracy: we expect the Lax synchronisation policy to be faster but less accurate than the SPEX. Although it is not our main objective, we also explore how well we can simulate targets with more cores than the ones available at the simulation host (i.e. $m > n$).

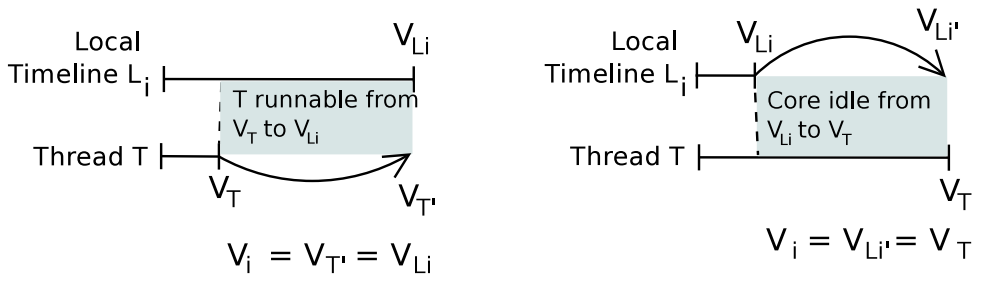


Figure 7.1: The two cases of resuming thread T on the local timeline L_i of virtual core i.

7.1 Methodology

7.1.1 Multiple core simulation

In the uniprocessor VEX presented in Chapter 3, all virtual time calculations and scheduling decisions take place on a single virtual timeline, which corresponds to the virtual timeline of a single processor. Since only one CPU resource is available, the simulation state is tightly coupled to the CPU state: if a thread is running then the single resource is busy and all other runnable threads must wait. If that thread is executing in time-scaled mode or simulating a model in virtual time, then all that happens is that the virtual time of the CPU is updated accordingly at the end of the thread's timeslice; the next runnable thread will resume exactly at that point.

We extend the uniprocessor VEX by defining a *local* virtual timeline for each of the m virtual cores of the simulation target. The purpose of the i-th *local* timeline L_i is to describe the progress of core i: if the current virtual timestamp of L_i is VL_i, then the next runnable thread T with current virtual timestamp VT_T, is resumed at virtual timestamp $V_i = max(VL_i, VT_T)$. The two possible cases are illustrated in Figure 7.1:

- If $VT_T < VL_i$, then the thread T can start executing at that core only after VL_i, as the resource was occupied by another thread until that point. From the virtual timestamp VT_T to VL_i the thread is supposed to have remained runnable in the virtual time execution.

- If $VT_T > VL_i <$, then when the resource was last released at VL_i, no thread was

immediately runnable, which only happened at VT_T. From the virtual timestamp VL_i to VT_T the core was idle in the virtual time execution.

Thread T executes on that core from V_i to $V_i + t_p$, where t_p is the virtual time progress of T (CPU time, I/O time etc), with $t_p \leq s$ and s the VEX scheduler timeslice. At the end of the timeslice, the virtual timestamp of the thread T is set to $V_i' = V_i + t_p$ and the thread is pushed back into the single priority queue of runnable threads, with priority V_i'. The thread with the lowest virtual timestamp is the head of the priority queue. Newly spawned threads inherit their virtual timestamps from their parents and are pushed onto the priority queue, waiting to be scheduled on the next available virtual core. Despite the fact that the timestamps of the threads are local, a global ordering of threads is still possible, as long as all local virtual timelines remain *in principle* synchronised. We elaborate on synchronisation schemes in the next section.

The previous VEX scheduler is now distributed to m VEX handlers (one per virtual core or local timeline), which are system threads that take the next runnable thread from a single queue, and schedule it on their own local virtual timeline L_i. This also entails synchronising with the thread that currently executes on their virtual core and suspending it when its timeslice s expires, i.e. after the handler thread sleeps for s. We clarify here that by s we refer to the timeslice of the scheduler of each virtual core. As a means to balance the load amongst the VEX handlers if multiple threads are spawned in a short period of time, a load balancer notifies virtual cores of the existence of a new thread in a round robin fashion.

7.1.2 Synchronisation

The need for synchronisation in multicore VEX deals with the problem that the m local virtual timelines may proceed at different paces, as they are in principle independent from each other. This might lead to causality errors, where events in the virtual time "future" (i.e. more than a scheduler timeslice ahead in virtual time) are executed before or in parallel with ones in the past, that might, however, affect their behaviour. In the context of virtual time execution

this matters even in the absence of direct communication points between threads, because VEX simulates and profiles the code on the actual host it is running and all threads share the resources of this system. This means that if the thread that is executing further ahead in virtual time issues a high load on the I/O subsystem, then this affects the measurements performed on the I/O operations of the threads being executed at the other virtual cores, at a former point in virtual time. The existence of shared memory access points might exacerbate the problem caused by the lack of synchronisation: as long as T can find an available virtual core to execute (in the virtual time "future"), it may update shared memory and notify other threads, exhibiting a behaviour that would not be possible in the correct virtual schedule (see Chapter 3).

This distinguishes multicore VEX from parallel simulation synchronisation techniques from execution-driven simulation that either synchronise at the shared memory level [113, 98], or the I/O level [7], or the network level [32, 14], or combinations of these [35, 21], in that there are no specified "interaction points" (which must be processed in order) between processes. We need that all processes progress at the pace determined by the VEX scheduler timeslice, in order to generate and profile the load on the simulation host (memory and I/O subsystem), as it would be observed if the virtual time schedule were realised. Our synchronisation assumption is thus that as long as the distance between any two local timelines is limited by one scheduler timeslice s, then the multicore simulation is considered satisfactorily synchronised.

A key observation in proposing a solution for causality errors in VEX is that the virtual time progress is tightly coupled with real time. This means that under the assumption that there exists one physical core for each virtual core ($m = n$), no other background system load is executing on the simulation host and threads run on the CPU without invoking any system calls (that are not accounted for by CPU-time counters), then the CPU-time progress within a virtual timeslice s is approximately equal to s (minus some time used by the VEX scheduler itself). As a consequence, we can assume that threads that execute on the m cores will progress at approximately the same pace and that in the general case the independent timelines progress in a synchronised manner without any additional effort.

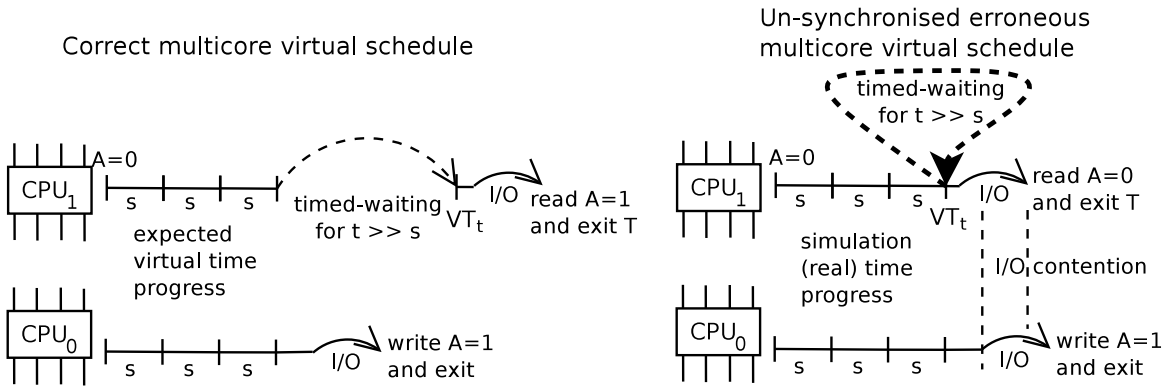


Figure 7.2: Demonstration of an erroneous simulation (wrong result and artificial I/O contention), due to lack of synchronisation in multicore VEX. The thread T executing at CPU_1 performs a virtual leap forward by t: without any synchronisation mechanism this violates the correct order of execution by progressing instantaneously the virtual time of CPU_1

Lax synchronisation mode

In the Lax policy threads are assumed to generally progress in virtual time in a synchronised manner without the need of explicit synchronisation interaction. Special handling is only added upon the occurence of specific events at which the virtual time progress deviates from the real one. Under the assumptions that $m = n$ and that there is no background load, then synchronisation handling only happens, when the currently running thread on a virtual core:

1. needs to perform a virtual leap forward
2. executes under the effect of a time-scaling factor or
3. simulates a performance model.

We elaborate on the mechanisms used in the Lax synchronisation mode to handle these three cases.

1. Virtual Leaps Forward: Imagine that T is "Timed-Waiting" (see Chapter 3) for a timeout t which expires at VT_T. Under the conditions about the state of the JVM elaborated in Chapter 5, VEX assumes that no other event can occur between now and VT_T and allows the virtual core i to leap forward to VT_T. This approach omits the fact that other threads might be executing on the remaining $m - 1$ cores, leading core i to execute at a virtual time

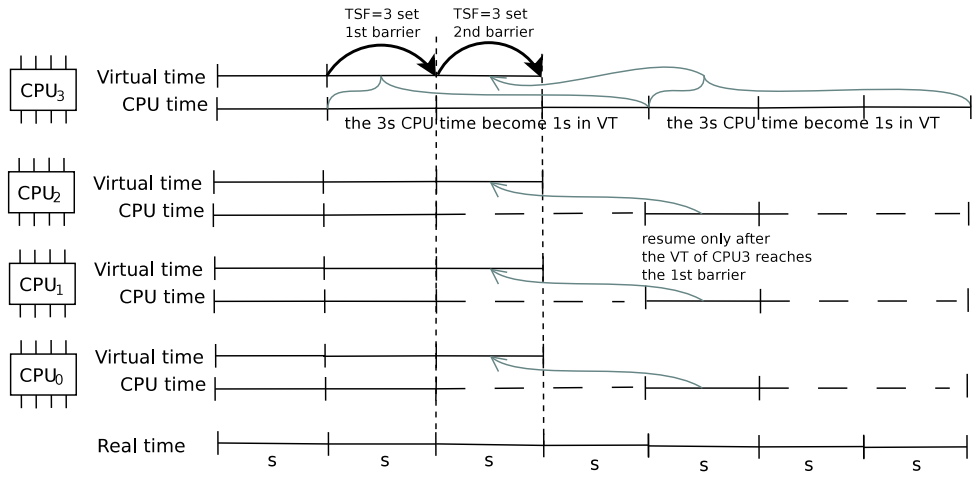


Figure 7.3: Time-scaling in Lax synchronisation policy on 4 cores. By using a barrier mechanism, the accelerated thread of CPU_3 executes 3 times as many timeslices as the threads of the other cores, while remaining synchronised with them in virtual time

arbitrarily far ahead of the other cores; this can lead to causality errors. The solution of the Lax synchronisation mode is to check whether the minimum virtual timestamp amongst all local timelines VT_{min} is within the distance of a scheduler timeslice s from the VT_T. If it is, then the thread is allowed to leap forward to VT_T, otherwise the thread handler of core i retries after sleeping in real time for $\min(s, VT_T - VT_{min})$.

2. Time-Scaling: By definition time-scaling of the execution of thread T creates a deviation between real and virtual time: assuming a time scaling factor TSF, then one virtual timeslice of duration s needs $(TSF \cdot s)$ real time to be simulated. If other threads are progressing without (or with a different) time scaling factor, then the synchronisation assumption between the timeline of T and the other timelines is broken. Assume that T is starting to execute a method with a speedup factor $TSF > 1$ and a current timestamp of VT_T. In this case the Lax synchronisation policy generates a barrier that will block other timelines until T progresses by the equivalent of one virtual timeslice. This barrier is set to the maximum virtual time VT_{max} that T can reach after completing its current virtual timeslice. When T reaches VT_{max} after $(TSF - 1) \cdot s$ real timeslices, the handler of the core where T is currently executing removes the barrier and the simulation progresses as normal. A simplified case of this is shown in Figure 7.3, where threads executing on virtual cores $0 - 2$ are stalled until the one in CPU_3 executes three times as much code in real time, thus simulating a TSF of 3. If $TSF < 1$, then we have a slowdown factor

and T reaches the end of its virtual timeslice before other threads do so in real time. In this case, all that the thread handler of this core needs to do after the thread completes its virtual timeslice, is to sleep in real time for up to s, thus allowing the other timelines to catch up.

3. Performance Model Simulation: The last case where the synchronisation assumption is broken happens when we are simulating performance models (see Chapter 8). Local virtual resources are simulated by moving the virtual timeline forward by the duration of (or the remainder of the current) scheduler timeslice s. Assuming that the execution time of this operation is small in comparison to s and that the cores were perfectly synchronised before the model simulation, then the virtual timestamp of the core where the model was simulated will be set ahead of the rest by s. To overcome this, we force the thread handler to sleep in real time for s, after updating a thread's timestamp according to the simulation of the model of a local virtual resource. This allows the other cores to make equivalent progress. If the local model exits within the current timeslice after $t_p < s$, then the handler thread only sleeps for the remaining timeslice $(s - t_p)$, allowing the thread to resume for the remainder of its timeslice. This approach resembles real time simulations [132, 39]. However, it only applies for local resources; being serviced by a remote resource is handled in the same way as virtual timeouts (waiting) are handled in virtual time. We also note that synchronisation points within the models are also dealt with in virtual time.

Despite its characterisation as Lax, this synchronisation scheme is conservative, in the sense that it does not allow progress in virtual time unless it is certain that the virtual execution order is maintained (within the distance of a scheduler timeslice s).

The Lax synchronisation was introduced under the assumption that, if no background load exists on the simulation host, then all cores progress in virtual time at the same pace (that of the real time). This assumption might not always hold, leading virtual cores to progress slower than others and lead to causality errors, even when none of the "special" cases that were described above arise. For example, if the background load on one of the cores is higher than the rest, then the OS scheduler might delay resuming, or assign less time to, the runnable thread

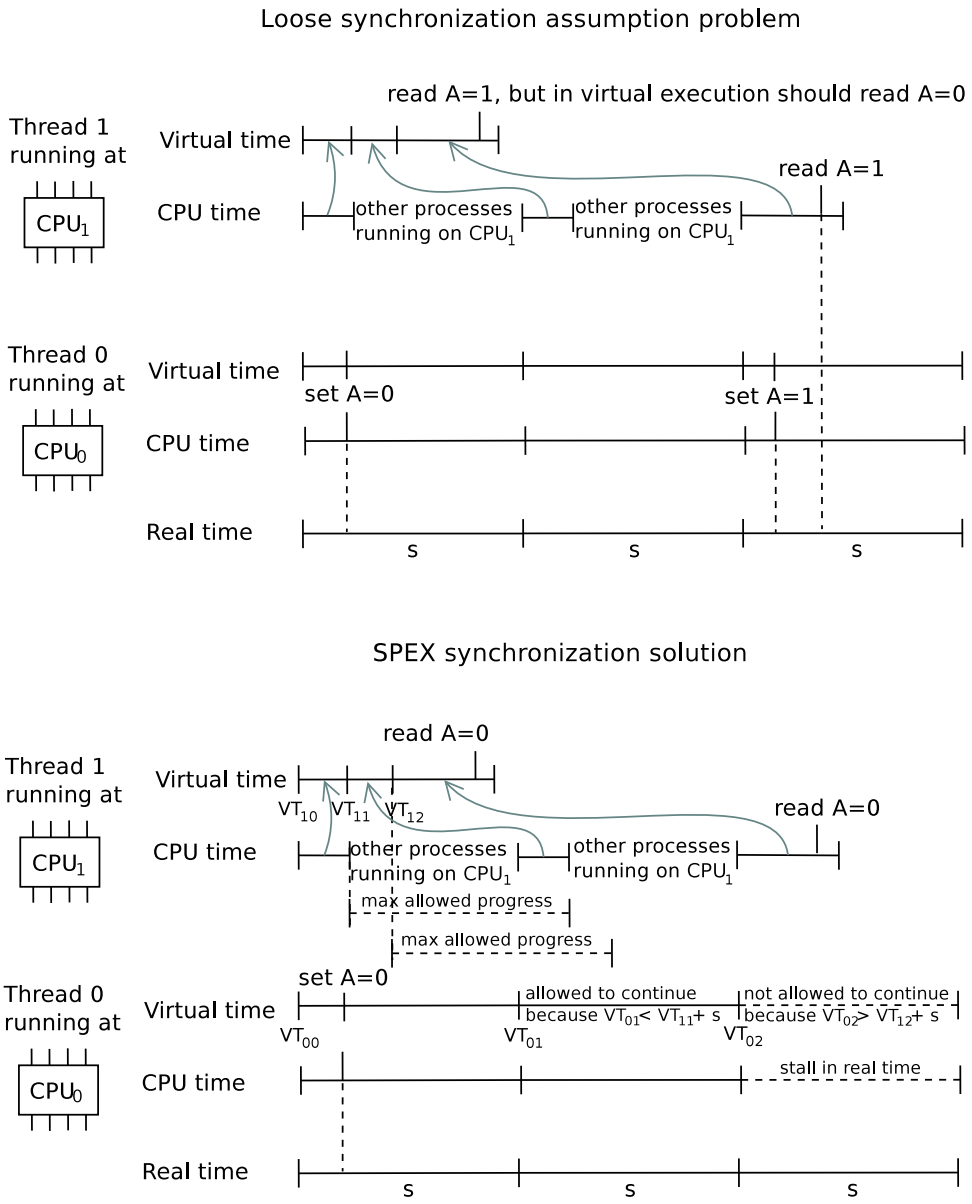


Figure 7.4: Problem with lack of synchronisation between two virtual cores on a dual-core machine with background load on CPU_1 and the solution of SPEX

that VEX set to running. This is illustrated in Figure 7.4, where the thread that is executing on CPU_1 progresses in virtual time slower than the one on CPU_0, due to the scheduling of other processes on CPU_1 by the OS. This leads thread 1 to read the updated value of the shared variable A that was set further ahead in virtual time (at the third virtual timeslice of thread 0).

Apart from this, if the CPU time of the thread is significantly lower than its real time (for example due to major page faults – see Section 3.3.3), or if the currently executing thread is blocked in uninstrumented code (see Section 5.2.2 for native waiting threads), then real

time progresses faster than CPU time for that core. A more subtle case that violates the Lax synchronisation scheme assumption that threads generally progress at the same pace, except from when specific events take place, occurs when the code executed by one thread on a virtual core is trapped to VEX (for example at method-profiling or state-changing instruments) at a higher rate than the threads on the other virtual cores. Since the CPU-time of the code inside the profiling component of VEX is not accounted in virtual time, its execution at a higher rate decelerates the progress of virtual time on the corresponding core. Nevertheless, we have already found (Section 3.3.2) that invoking instrumentation code at a high rate should be avoided through the use of adaptive profiling schemes. These adaptive profiling policies would then also be expected to alleviate the aforementioned effects on inter-core synchronisation.

Strict parallel execution (SPEX)

To overcome these issues we introduce an alternative *strict*-parallel execution (SPEX) mode, which explicitly enforces a synchronisation mechanism, so that all local timelines progress together, with a maximum deviation of a single timeslice s. We separate cores into "active" and "disabled", meaning "busy" and "idle" respectively: only the local timelines of active cores are considered in each synchronisation step. If no runnable thread exists to be scheduled on a previously active core, then the core is disabled.

Using these two notions, we implement SPEX by defining a global virtual time (GVT), which is the smallest local virtual timestamp amongst all *active* cores. GVT is updated every time a thread is about to change its state. Recall from Section 7.1.1 that whenever a thread handler is about to resume a previously suspended thread T with virtual timestamp VT_T at core i with local virtual timestamp VL_i, then it will resume it at a timestamp $VT'_T = max(VL_i, VT_T)$. In this case, assuming a scheduler timeslice of s, the following rules are enforced:

- If $VT'_T - GVT \leq s$, then the VT'_T is less than a scheduler timeslice ahead of the GVT, which means that allowing the thread to resume does not violate our synchronisation assumption. Accordingly, the virtual core is enabled and the thread is resumed with virtual timestamp VT'_T.

- If $VT'_T - GVT > s$ and another active core exists, then the VT'_T is more than a scheduler timeslice ahead of the GVT and allowing the thread to be resumed would violate the synchronisation assumption. Therefore, the core is disabled until the other active core(s) catch up.

- If $VT'_T - GVT > s$ and all cores are disabled, then even though the VT'_T is more than a scheduler timeslice ahead of the GVT, there is no other activity in the simulation (as all other cores are idle) and so the thread is allowed to resume with virtual timestamp VT'_T.

The above rules ensure that no two threads become separated in virtual time by more than s. Figure 7.4 illustrates this: CPU_0 becomes disabled at VT_{02}, because the committed time of CPU_1 up to that point VT_{12} is more than a timeslice s smaller than VT_{02}; this is due to background load, for example. We note that the same policy is also enforced when a thread handler decides on whether to continue executing a currently running thread on its virtual core (in lack of any suspended runnable threads). Based on the distance of the virtual timeline of the core from the GVT, the handler may force such a thread to become suspended, though no other reason exists for the thread to context switch.

Using these rules SPEX also covers the cases that require special handling in the Lax synchronisation scheme, i.e. virtual leaps forward, time scaling and model simulation. A virtual leap forward $VT_T > GVT + s$ is only allowed, when there are no other runnable threads and all cores are disabled. This offers a simple solution to the multicore synchronisation problem, but adds an overhead of at most $(m-1) \cdot s$ in real time, when all but one of the cores have to be successively disabled (and stall) to allow leaps that are longer than s to take place. Time-scaled threads that move faster in virtual time ($TSF < 1$) are halted, until the remaining running threads reach them. Finally, the simulation of a model is resumed only if it does not violate the rules, i.e. if the previous update of the VT_T due to a model simulation is not too far ahead in virtual time. Otherwise, the next update is stalled.

The strict parallel approach constitutes essentially an adaptation of the conservative time window synchronisation [82] with the distance between two parallel processes (here threads executed on virtual cores in parallel) is bound by the VEX scheduler timeslice s.

7.2 Extensions for VEX thread handling

We have seen how multicore VEX handles timed-waiting, time-scaling and model-simulation for both the Lax and SPEX synchronisation schemes. In this section we elaborate on the I/O and native waiting thread states.

7.2.1 I/O

I/O operations are issued according to the order defined by the virtual time schedule, but are profiled in real time on the host's I/O subsystem (see Chapter 4). This imposes a dependence between the effectiveness of the synchronisation scheme used and the correctness in terms of likeliness of the virtual schedule happening for real (see Figure 7.2). Assuming that the cores are synchronised, then the I/O handling component of VEX works as in the uniprocessor case: when a thread is about to perform an I/O prediction (scaled or normal), then the measurement buffers that are used for the learning and prediction phases and that are shared amongst all application threads are used. The rules for the correct virtual time scheduling in light of "Standard I/O" operations (see Section 4.1.2), also apply in the multicore case without any modifications. This is attributed to the fact that all cores share the same queue of runnable threads: pushing onto the queue a record with a timestamp equal to PVT is then able to control which threads should progress in parallel.

The completion of the I/O operation needs to be identified as soon as the operation finishes, to reschedule the issuing thread and simulate priority boosting. In the uniprocessor case the VEX scheduler reduces its timeslice and monitors the progress of the I/O predicting thread at a higher rate (see Section 4.1.2). In multicore VEX this responsibility falls to the handler of the last virtual core, on which the thread was scheduled, before issuing the I/O operation. The thread that completes its I/O prediction pushes itself back to the queue of runnable threads, after updating its virtual timestamp according to the observed I/O duration. Either the reduced-timeslice handler or any other will find the thread on the queue and resume it, after updating its time according to their own timelines and the thread's virtual timestamp (see Figure 7.1).

When no more threads are in an I/O-related state (learning or predictive period), all handlers reset their timeslices to the defaults.

7.2.2 Native waiting

The problem with native waiting threads (see Section 5.2.2) is that we cannot know when they start and when they finish. Using counter- or stack-trace-based techniques, we identify in the uniprocessor VEX that a thread has become native waiting at some point in its last timeslice (Section 3.1.2). In multicore VEX this is accomplished by the virtual core handlers.

After being recognised as native waiting at time t_0, a thread is allowed to execute uncontrolled, until it invokes a method of the VEX framework. At this point the thread returns under VEX's control and the simulation resumes normally. However, the CPU time that was accumulated until that point by the supposedly natively blocked thread needs to be accounted for. In the uniprocessor case this duration t_{nw} was assigned to the single CPU and was mapped to the virtual timeline taking into account the time progress of the threads that were executed concurrently with it. In the multicore setting we add t_{nw} to the core whose virtual timeline has the lowest virtual timestamp. The assumption is that the minimum local virtual time VL_i implies that the virtual core i was idle when the supposedly "native-waiting" thread was making progress.

An example that demonstrates the reasoning behind this decision is given for $m = 2$ virtual cores. We have the following two cases:

- If core 0 has been idle and core 1 has been busy since t_0, then the local times would be $VL_0 < VL_1$ and the time t_{nw} would be "correctly" mapped to core 0.

- If both cores have been busy since t_0, then $VL_0 \approx VL_1$ and the duration t_{nw} could essentially be mapped to any of them. This would lead to a gap between the two cores, that could only be handled by the strict synchronisation scheme.

In any case we see that mapping the time to the virtual core with the minimum local virtual time, yields a "correct" virtual time schedule and one which is arguably reasonable.

7.3 Evaluation

7.3.1 JGF investigation of scaling

We first validate the multicore virtual time approach by executing the multi-threaded JGF benchmarks [68], whose uniprocessor results have been evaluated in Chapter 6.3. First we execute the tests on the dual-core ($n = 2$) Host-1, as described in Appendix A and then we simulate them with VEX on the same host on two virtual cores ($m = 2$). The scheduler timeslice is set to $s = 10ms$. We evaluate our approach based on how closely the observed real time results match the ones returned by the virtual time simulation.

These results for the JGF benchmarks are presented in Table 7.1. The geometric mean of the prediction error of multicore VEX (with Lax synchronisation) is found to be 6.94%, while the mean simulation overhead is 1.65. As with the uniprocessor simulation the workload size affects the prediction accuracy. The outliers ($MonteCarlo$, $RayTracer$ and $LUFact$) remain the benchmarks that invoke the `Thread.yield()` or JIT-compile different methods in the RTE and the VTE. The only exception is the SOR benchmark whose prediction error and overhead are increased. The reason lies in the busy-waiting synchronisation that this benchmark uses which, in fact, produces variable results even in the real time execution (according to OS scheduler decisions).

Overall, the simulation of the JGF benchmarks on a dual-core virtual host does not significantly increase the prediction error or simulation overhead. Up to this point, we have only compared the observed results of the RTE to the predicted virtual times of the VTE. A different way to investigate the effectiveness of the multicore VTE is to determine how well it reflects the scalability of each benchmark. The last three columns of Table 7.1 show the ratio of the result of the dual-core RTE to that of the uniprocessor RTE and the corresponding ratio in virtual time

Threads: 8			Real Time		Multicore VEX Lax (10ms)				Scaling vs 1 core		
Size	**Benchmark**	**Methods**	**Result**	**Time**	**Result**	**Time**	**Error%**	**O/h**	**Real**	**VEX**	**Diff.%**
A	Crypt	82	0.202	0.322	0.146	0.377	-27.72	1.22	2.38	1.51	-36.59
A	LUFact	188482	0.189	0.347	0.236	2.086	24.87	6.12	1.48	1.19	-19.92
A	Moldyn	407503	2.117	2.226	1.755	3.437	-17.10	1.77	1.88	1.37	-26.96
A	MonteCarlo	923971	1.833	2.289	1.821	4.426	-0.65	3.06	1.78	1.57	-11.96
A	RayTracer	1133320	1.261	1.366	2.172	7.079	72.24	5.89	1.90	1.32	-30.53
A	Series	30046	4.422	4.52	4.377	5.195	-1.02	1.15	1.93	1.85	-4.17
A	SOR	48	2.57	4.116	3.653	5.843	42.14	1.39	3.39	2.72	-19.78
A	SparseMatMult	47	0.201	0.447	0.172	0.523	-14.43	1.21	1.89	4.88	158.32
	Geomean-A:						11.67	2.15			23.52
B	Crypt	82	0.799	0.999	0.737	1.195	-7.76	1.08	2.12	2.16	2.00
B	LUFact	220239	1.187	1.365	1.176	3.288	-0.93	2.94	1.30	1.16	-10.21
B	Moldyn	458161	18.995	19.09	16.227	20.83	-14.57	1.36	1.48	1.50	1.13
B	MonteCarlo	3357148	9.271	10.135	9.141	17.032	-1.40	2.73	1.74	1.67	-4.17
B	RayTracer	1299816	11.143	11.258	19.947	19.619	79.01	1.87	1.76	1.46	-17.24
B	Series	95659	45.051	45.145	45.168	48.396	0.26	1.10	1.93	1.83	-4.84
B	SOR	48	3.085	4.706	4.012	6.084	30.05	1.37	2.99	2.52	-15.86
B	SparseMatMult	47	0.5	0.814	0.446	0.881	-10.80	1.10	1.30	2.02	55.23
	Geomean-B:						5.61	1.57			7.18
C	Crypt	82	1.851	2.184	1.784	2.557	-3.62	1.09	2.07	1.97	-4.89
C	LUFact	257735	10.417	10.791	8.221	12.523	-21.08	1.35	1.08	1.16	7.04
C	Series	75849	696.62	697.87	701.51	750.16	0.70	1.08	2.11	1.96	-7.13
C	SOR	48	4.451	6.178	4.662	7.034	4.74	1.30	2.25	2.28	1.38
C	SparseMatMult	47	8.75	9.498	8.267	9.692	-5.52	1.07	1.25	1.35	7.78
	Geomean-C:						4.26	1.17			4.83
	Geomean						6.94	1.65			10.26

Table 7.1: Multicore VEX predictions for the multi-threaded JGF benchmarks executed by 8 threads on 2 (physical and virtual) cores. Scaling results are compared to the predictions of uniprocessor VEX presented in Table 6.4. Means of 10 runs are shown.

predictions. This is interesting, because it allows us to evaluate how effectively the multicore virtual time execution can be used for investigating the scalability of an application with an increasing number of cores.

Although the prediction accuracy improves between the smallest and the largest workload set, the results are inconclusive, in that large differences exist in all groups. Specifically, the short running time of the dual-core *SparseMatMult* (which suffers from JIT effects) and the aforementioned busy-waiting of *SOR* create large variations between the RTE and the VTE scaling. However, the dual-core scalability of long running JGF benchmarks like *MonteCarlo-B* and *Series* that do not exhibit behaviours like synchronisation with busy-waiting or `Thread.yield()` leads to close matches between the real and the virtual time execution.

7.3.2 SPEC investigation

Next we use the multicore VEX to predict the performance of the SPECjvm2008 [128], that we have extensively investigated in Section 6.4. By enabling both cores of Host-3, we acquire

the real time throughputs for each benchmark and measure the benchmark execution times, when executing it for 24s of warmup and 96s of main iteration time. Then we perform the same experiment in virtual time, by leaving both cores of the (simulation) host enabled ($n = 2$ physical cores) and setting VEX to simulate $m = 2$ virtual cores.

Threads: 4	**Real time**		**Lax sync 10ms on 2 cores**				**SPEX 10ms on 2 cores**			
Benchmark	**ops/m**	**Time**	**ops/m**	**Time**	**Err.%**	**O/h**	**ops/m**	**Time**	**Err.%**	**O/h**
comp.compiler	175.92	131.92	195.84	160.13	11.32	1.21	223.53	190.18	27.06	1.44
comp.sunflow	80.45	133.58	89.15	161.84	10.81	1.21	91.62	173.76	13.88	1.30
compress	83.73	128.10	82.40	136.27	-1.59	1.06	86.12	141.07	2.85	1.10
crypto.aes	26.13	144.43	25.80	152.99	-1.26	1.06	27.46	162.58	5.09	1.13
crypto.rsa	131.93	126.04	114.67	130.35	-13.08	1.03	132.08	141.39	0.11	1.12
crypto.signverify	140.92	125.30	144.34	176.43	2.43	1.41	149.29	180.63	5.94	1.44
derby	90.20	267.54	77.52	352.00	-14.06	1.32	83.39	419.26	-7.55	1.57
mpegaudio	44.78	135.45	47.76	176.71	6.65	1.30	48.07	203.94	7.35	1.51
sci.fft.large	10.95	187.15	11.43	189.09	4.34	1.01	11.53	197.00	5.25	1.05
sci.fft.small	122.10	126.22	130.53	209.03	6.90	1.66	132.55	187.04	8.56	1.48
sci.lu.large	3.31	378.32	3.44	403.53	3.78	1.07	3.67	361.38	10.88	0.96
sci.lu.small	148.03	124.64	155.31	159.36	4.92	1.28	158.14	151.26	6.83	1.21
sci.monte_carlo	83.30	127.35	91.10	134.03	9.36	1.05	91.96	139.69	10.40	1.10
sci.sor.large	15.66	164.82	16.41	178.49	4.79	1.08	16.54	175.36	5.62	1.06
sci.sor.small	88.08	128.03	90.40	133.18	2.63	1.04	90.62	136.41	2.88	1.07
sci.sparse.large	10.31	186.46	11.63	207.28	12.80	1.11	11.98	216.34	16.20	1.16
sci.sparse.small	65.20	129.38	62.69	156.72	-3.85	1.21	70.17	158.72	7.62	1.23
serial	57.61	132.19	61.46	145.11	6.68	1.10	63.32	155.13	9.91	1.17
sunflow	28.35	142.14	27.72	281.08	-2.22	1.98	27.77	305.61	-2.05	2.15
xml.transform	121.77	145.08	120.58	206.36	-0.98	1.42	130.42	209.60	7.10	1.44
xml.validation	169.92	124.18	180.52	143.12	6.24	1.15	182.07	152.11	7.15	1.22
Geo-mean					4.81	1.21			5.89	1.26
Mean					6.22	1.23			8.11	1.28

Table 7.2: Multicore SPECjvm2008 predictions for 4 threads on 2 (physical and virtual) cores. Means of 3 runs are shown.

The results of the SPECjvm2008 for 4 threads executing on the dual-core host with a 10ms timeslice are presented in Table 7.2. Lax is slightly more accurate than SPEX with a geometric mean of prediction error of 4.81% (compared to SPEX's 5.89%). As expected, SPEX yields a higher simulation overhead (with a geometric mean of 1.26 compared to Lax's 1.21), due to the delays added to synchronise the timelines of the virtual cores.

We observe that SPEX predicts higher throughputs than Lax for all benchmarks. This is attributed to an unrealistic virtual time schedule that is observed in light of native waiting threads in the Lax synchronisation policy. Specifically, assume that a native waiting thread commits the CPU-time progress Δt that it made while remaining uncontrolled by VEX. Δt is then added to the timeline of the virtual core i with the minimum virtual time VL_{min} (see Section 7.2.2). If the difference of VL_{min} to the other timelines is small and Δt is high (i.e. greater than a scheduler timeslice s), then the virtual timeline i will "leap" for up to Δt time

further ahead in the future than any other core j with $j \neq i, j = 0...m-1$. In SPEX, this leads to i being stalled (disabled) until the other cores synchronise with it. In Lax, however, the simulation will continue normally, as this "leap" is not recognised as an event that requires any special handling. What happens then is that any threads T_i that are scheduled on core i will resume at the virtual timestamp of that core, VL_i, which is higher than the timestamps of the other cores. When these threads T_i become suspended and pushed onto the queue of runnable threads, their virtual timestamp will thus be higher than the one of the threads that are executing on any other virtual core, because, after the "leap" $VL_i > VL_j$ for any $j \neq i$. Threads T_i will only be scheduled on i, whilst the other $m-1$ virtual cores execute exactly $m-1$ threads with lower timestamps (assuming such threads exist). Eventually, this leads to the same $m-1$ threads executing on the virtual cores with lower virtual timestamps than i and all other threads executing in a round robin fashion on core i. This situation is only resolved, when the other cores reach the virtual timestamp of i.

This effect results into an unrealistic and imbalanced virtual time schedule, which lowers the predicted throughput of the Lax method, as $m-1$ threads are essentially bound on a single core and all other threads contend for the other one. In contrast to this, SPEX allows a round robin execution, even when a virtual core is stalled. Interestingly, it is Lax that offers the most accurate predictions; the strict parallel round robin schedule of SPEX yields higher throughput results. We believe that though both synchronisation methods disregard the effect of background load and system threads, Lax's unusual mode of operation in light of native waiting threads, indirectly compensates for this, thus returning more accurate results.

7.3.3 Simulating a large number of CPUs

Up to this point our focus is on simulating the behaviour of an application on a specific host, with the same number of the virtual cores in the multicore VTE as the simulation host, i.e. $m = n$. In this section we explore how well our synchronisation methods cope with an increasing number of virtual cores m to up to 64 on a dual-core machine $n = 2$ (Host-3 of Appendix A). The case where $m > n$ is tantamount to exploring the application's behaviour on a *different*

host. However, as it is straightforward to do we validate this using a contrived toy example, which consists of a floating point calculation iterated for a number of times within a loop. Our focus is on the correctness of the virtual time accounting scheme and the differences between the synchronisation schemes for normal and time-scaled executions.

The execution time of the loop was profiled to be approximately $q = 170ms$ on a uniprocessor. The loop is CPU-bound. No caching effects are exhibited as all operands of the calculated expression fit in CPU registers and there is no interaction between the T threads executing the loop in "parallel" on the m virtual cores. This allows us to assume that the model $f(T, m) = q\lceil\frac{T}{m}\rceil$ returns the total execution time as a function of T and m. The idea is that in such a simple and scalable program, any deviations from $f(T, m)$ must be attributed to the simulator and not to any other system factors.

Figure 7.5a shows the VEX-predicted times of our program $v_l(T, m)$ as T threads (from 16 to 256) are executed on an increasing number of m Lax-synchronised virtual cores. The results confirm our expectations of low prediction errors in such trivial CPU-bound cases, since they are close to $f(T, m)$ with a geometric mean of 5.33% amongst all sample (and arithmetic mean of 10.39%).

Analysing the results we find that the lower the ratio $\frac{T}{m}$ the higher the prediction errors (related to $f(m)$) (see Table 7.3). This is attributed to lack of synchronisation between the cores in the

		Lax		SPEX	
T / m	f(T, m)	Prediction [s]	Error %	Prediction [s]	Error %
1	0.17	0.19	8.99	0.18	6.29
2	0.34	0.49	44.08	0.36	6.75
4	0.68	0.76	11.06	0.70	2.64
8	1.36	1.46	7.28	1.37	0.96
16	2.72	2.81	3.17	2.70	-0.87
32	5.44	5.51	1.20	5.35	-1.61
64	10.88	10.80	-0.73	10.64	-2.17
128	21.76	21.19	-2.62	21.10	-3.04
256	43.52	41.24	-5.23	41.36	-4.95
Geomean			5.33		1.84
Mean			10.39		3.01

Table 7.3: Multicore VEX predictions for the ratio $\frac{T}{m}$ with T threads and m virtual cores

Lax synchronisation scheme. Specifically, for $\frac{T}{m} = 1$, the load balancer assigns one thread to

each virtual core and each thread runs to completion uninterrupted. This is not the case for higher values of T, where OS scheduling might lead a core to execute more threads than its $\frac{T}{m}$ share. This happens if a thread completes at a core and another runnable thread is immediately assigned to it, though there could have been another virtual core that was available at a former point in virtual time. Since we measure the time that all threads require to complete their loops and join with "main" (that spawned them), even a single core making progress faster and executing a higher load than it should, increases the estimated application response time.

We overcome this by enabling the SPEX mode (results demonstrated in Figure 7.5b): in this case a core that is further ahead in virtual time awaits until the others reach it. This yields more accurate results, because threads are only resumed when they don't violate the s timeslice barrier (Table 7.3). Table 7.4 shows that the simulation times of the two synchronisation modes without any time-scaling ("Normal" columns) are very close, with Lax synchronisation being marginally (3%) faster than SPEX.

Next we use our toy program to investigate the effects of time-scaling. The effects of the scaling factor $TSF = 0.5$ are theoretically reflected in the expected running time of the program via the function $f'(T, m) = \frac{f(T,m)}{TSF}$.

Parameters			**Lax**			**SPEX**			**SPEX/Lax**	
			TSF 0.5			**TSF 0.5**				
m	T	f'	**Res.**	**Time**	**Err.%**	**Res.**	**Time**	**Err.%**	**Norm**	**TSF**
4	16	0.34	0.38	3.94	11.57	0.34	3.92	0	1.15	0.99
4	32	0.68	0.73	5.46	7.61	0.68	5.4	0	1.01	0.99
4	64	1.36	1.44	30.33	6.19	1.36	8.5	0	1.04	0.28
4	128	2.72	2.91	56.42	7.04	2.69	14.39	-1.1	1.02	0.26
4	256	5.44	5.6	62.41	3.03	5.4	27.59	-0.74	1.03	0.44
8	16	0.17	0.2	3.9	15.81	0.18	3.95	5.88	0.99	1.01
8	32	0.34	0.36	5.33	5.75	0.35	5.36	2.94	0.98	1.01
8	64	0.68	0.71	14.77	3.96	0.68	8.33	0	1.03	0.56
8	128	1.36	1.4	36.63	3.07	1.36	14.05	0	1.01	0.38
8	256	2.72	2.78	60.91	2.14	2.71	25.38	-0.37	1.05	0.42
16	16	0.09	0.09	3.88	6.4	0.1	3.94	17.65	1.04	1.02
16	32	0.17	0.19	5.37	10.45	0.18	5.3	5.88	1	0.99
16	64	0.34	0.38	26.29	13.15	0.35	8.39	2.94	0.99	0.32
16	128	0.68	0.7	19.72	3.28	0.68	14.04	0	1.02	0.71
16	256	1.36	1.39	38.67	2.36	1.36	25.67	0	1.07	0.66
32	16	0.09	0.09	3.95	10.17	0.09	3.91	5.88	1	0.99
32	32	0.09	0.1	5.29	15.95	0.09	5.34	5.88	0.94	1.01
32	64	0.17	0.19	13.71	10.85	0.19	8.3	11.76	1.07	0.61
32	128	0.34	0.38	29.79	12.32	0.35	14.05	2.94	0.99	0.47
32	256	0.68	0.71	33.92	4.75	0.69	25.57	1.47	1.18	0.75
Geomean					6.52			0.19	1.03	0.63
Mean					7.79			3.05	1.03	0.69

Table 7.4: Multicore VEX predictions and overheads for "Normal" and time-scaled (accelerated by 50%) executions of T threads on m virtual cores

The results under the "TSF 0.5" columns of Table 7.4 show that both synchronisation schemes provide predictions for f' within 6.52% error. Between them, SPEX performs much better, with a geometric mean of the absolute prediction error compared to f' (column f' Error%) of less than 0.5%. An interesting observation is that, when TSFs are defined, SPEX significantly outperforms Lax. This is despite the fact that the overheads of the SPEX and Lax synchronisation schemes in the unscaled VTE are approximately the same (last columns of Table 7.4) with Lax being marginally faster by 3%. Clearly, the barrier of the Lax synchronisation scheme does not scale well with increasing numbers of threads and virtual cores, as each barrier is more likely to block one or more otherwise runnable threads from making progress. This forces the virtual core handler to sleep and retry.

We note here that in the general case of simulating more complex applications, than the presented one, with $m > n$, we would not expect the multicore virtual time execution to be accurate. This is because the effects on the simulation host (TLB misses, cache misses, etc.) as a result of the contention of the m virtual cores on the n physical ones affects our profiling measurements and/or creates synchronisation deviations in the Lax policy. On the other hand, cache invalidation costs or cache-to-cache transfers and thread migrations that may affect stall cycles on a core [75] and that *could be* observed with m real cores are not captured by multicore VEX. Predicting such effects would require knowledge of cache levels, cache-sharing and sizes of the virtual cores and instrumentation of the access patterns of the application under test. Recall that VEX's objective is to offer high-level lightweight performance predictions, without resolving to full system simulation [84, 90, 149]; instead it models processing resources by performing measurements on the existing ones. Although the latter might be affected by the existence of VEX (in the uniprocessor simulation) or running threads of other virtual cores in the multicore version, reduced accuracy is the price we expect to pay for keeping the simulation overhead low.

In conclusion, we simulated two sets of benchmarks on a dual-core host for $m = n = 2$, relying on the OS scheduler to map the threads that VEX defined as running to the underlying physical cores. The conclusions from this study are:

- The multicore VEX provides fairly accurate but variable results for the JGF multithreaded benchmarks for reasons similar to the uniprocessor case, like JIT effects, synchronisation with *Thread.yield*, busy-waiting etc.

- The SPEX and Lax synchronisation policies can accurately predict the throughput of the SPECjvm2008 benchmarks at a slightly higher overhead than the uniprocessor case.

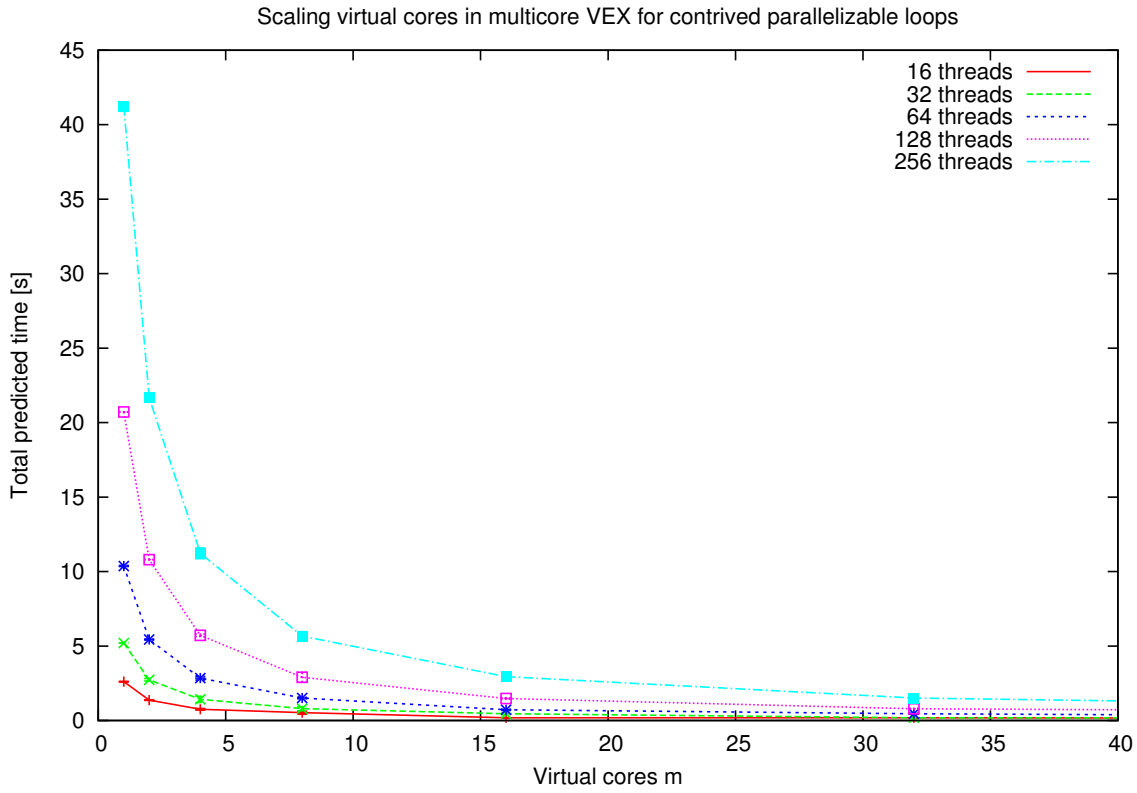
- In the SPECjvm2008 benchmarks the Lax synchronisation has been found to be more accurate than SPEX, which we attribute to its less intrusive nature and the existence of native waiting threads and safepoint synchronisations.

We also experimented with a contrived program for the case that $m > n$ and found that:

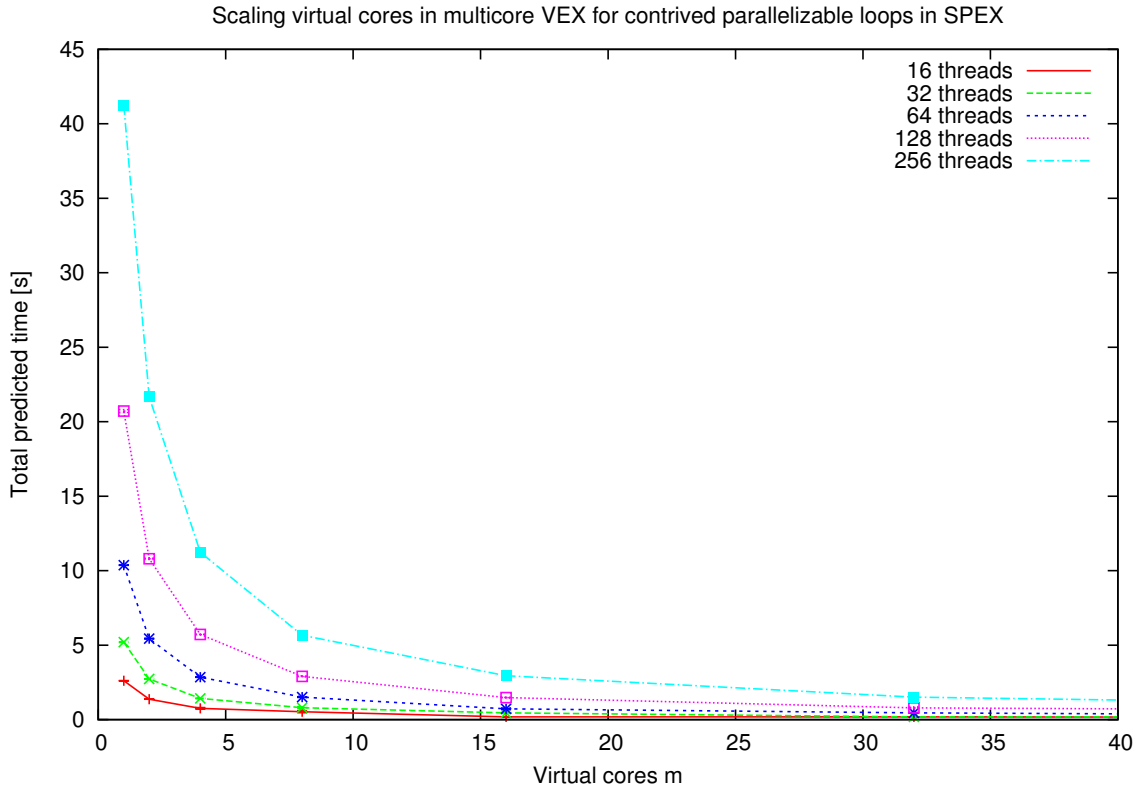
- SPEX yields better predictions in any case due to its independence from OS scheduling decisions

- When $\frac{T}{m} \approx 1$ and $m >> n$, the Lax synchronisation schem might lead to a virtual core being assigned a higher load than it should, due to OS scheduler decisions.

- The overheads of SPEX and Lax are similar in an unscaled VTE, but SPEX is significantly faster when TSFs are defined.



(a) Lax synchronisation



(b) SPEX

Figure 7.5: Scalability of multicore VEX in terms of virtual cores m on a toy example with simple loops with approximately 95% confidence intervals (3 runs)

Chapter 8

Model integration

In this chapter we present an extension to the VEX framework to support continuous, lifecycle-wide performance testing. The main idea is to integrate performance models for specified parts of an application with measured execution times from those parts for which there is existing, instrumentable, code. The performance models can be used to provide alternative timing estimates for existing code or for code that has yet to be written. The ability to mix real code and performance models seamlessly allows us to test performance metrics from the inception to completion of a software development project. Typically, this process will begin with a skeleton application comprising mostly, if not exclusively, performance models; at the end of the cycle we will have a complete application comprising 'production' code. VEX enables the virtual execution times from the simulation of models to be mapped together with real execution-time measurements onto a *global virtual timeline*. In principle, any formalism can be used to define a performance model; here we will focus on (open) queueing networks that are specified using the Java Modelling Tools (JMT) [16] and simulated in virtual time using a separate discrete-event simulation tool. The virtual time spent in a queueing network corresponds to a particular method/function call at the application level and this time is used by VEX in part to control the thread schedule. Importantly, different methods can invoke the same model by injecting virtual jobs into it, typically at different virtual times. If the model has no state (for example a simple infinite server node), then the model-predicted time delay can be determined instantly.

In general, however, the time spent in the queueing network will be dependent on the network's current state and possibly also future virtual jobs arrivals (if there is overtaking, for instance).

A key idea to this approach is that the contention on resources described by models is also taken into account when executing the real threads of the application. Models are simulated, while real-code is executed in a normal, or possibly time-scaled, manner. The simulation models resource consumption and queueing, so that a thread that is awaiting a virtual resource in the model is suspended waiting for the resource to be released. VEX provides a fair schedule of threads in virtual time, regardless of whether they are executing real code or simulating performance models. VEX updates the virtual time of a model-simulating thread instantly and still maintains a valid execution order by deciding which thread is to be scheduled next. This means that models and code can be used interchangeably.

This approach also allows us to use modelling in post-implementation stages to investigate hypothetical scenarios, by associating performance models with methods whose code is already implemented. In that case, the method is executed normally, but the execution time logged onto the virtual timeline is provided by a model. This decouples the functional and non-functional behaviour of the code, much like traditional execution-driven simulation approaches presented in Section 2.2.1.

Up to this point VEX models the time-scaling of a method's execution time by allowing a thread to progress at a different pace than others in virtual time. We now extend VEX's modelling capabilities, by entirely decoupling the duration of the thread execution on the simulation host from its virtual-time performance. Instead, its expected performance is now defined by simulating a performance model in virtual-time. The immediate effect is that the method body is no longer needed to determine the virtual-time performance, which allows us to investigate hypothetical performance scenarios throughout the lifecycle, for example using stubs and mock objects [46].

Entire subsystems may be investigated in the same way. For instance VEX can model the behaviour of an alternative I/O subsystem and identify the effect that its deployment would have on the application performance. I/O requests are executed as normal in real time. When

many requests are issued concurrently, they are simulating the same subsystem model, thus contending for its resources. Their sequence and their execution times are defined in the virtual timesphere of VEX.

This chapter discusses the methodology for integrating performance models and real code measurements in virtual time to enable end-to-end performance testing throughout the software development lifecycle. Details are provided on the extensions to the VEX framework and the JMT suite to support the integration of queueing networks and executing code. We demonstrate the performance engineering methodology that VEX facilitates by documenting a sample scenario – the development of a client-server application based on a MySQL database.

8.1 Integration of models and code

Performance models and implemented programs are used differently in performance analysis. Performance models yield predictions by simulating statistically described workloads in virtual time, abstracting away implementation details. In contrast to this, programs execute the designated workloads on a particular system so the measured performance reflects the actual consumption of resources. To enable the co-existence of models and code for performance analysis, we need to reconcile their differences.

8.1.1 Merging the notion of time

The notion of time in the execution of programs is typically related to observed wall-clock *real times*, calculated as machine clock cycle counts divided by clocks per second. All system counters are related to accumulated clock cycle counts, thus progressing "continuously" in time at the granularity of a single clock cycle. In a performance model time is virtual and in a simulation model in particular time is advanced instantaneously from one scheduled event to the next.

Incorporating real time observations to simulation times [125] or vice versa [39], may either

abstract away implementation details or increase the simulation time. The two notions of time are merged by being mapped onto a single virtual time line within the VEX framework. The role of the VEX scheduler remains to control the progress of the program threads by maintaining timestamps for each of them (executing real code or simulating models), within a timeslice distance as explained in Section 3.1.2. For example, if an event is scheduled to happen Δt in the future according to some model, then existing threads are assigned Δt of execution time, before the event is triggered. Only if no other runnable threads exist at the time, is the framework allowed to advance the virtual time by Δt. By controlling the execution of threads based on thread-local (virtual) time stamps, VEX maintains a virtual time schedule that is consistent with both code execution and performance model simulation.

VEX is also in control of virtual time logging, thus being able to select which execution times are to be included in its measurements (see Section 3.2). Up to this point this allowed us to isolate the overhead of the simulation framework itself and disregard it in the profiling results. With the model integration approach, the same feature enables us to replace the measured execution time of an implemented method with a (virtual) time computed by a model. This may be desirable in a "what if" experiment aimed at determining the sensitivity of some performance measure to the execution time of a particular method.

8.1.2 Merging workloads

When a thread T invokes a method whose execution time is described by a performance model, we trigger a new job arrival event in the model simulation as shown in Figure 8.1. The VEX scheduler then simulates the progress of the job within the model, whilst changing VEX's internal state associated with T. The overall effect is that the other threads proceed in virtual time using the state of T as defined by the performance model.

From the operating system perspective, thread T remains runnable. This means that T proceeds to execute the body of the model-simulated method (thus implementing the correct functional behaviour) whilst the VEX scheduler controls the scheduling of events within the model. Figure 8.1 shows an example execution where a thread T executes a method M in real

time, whilst the time VEX uses to advance T's virtual clock is determined by a performance model.

Both T, and its associated virtual job in the queueing network, synchronise at the point where both the method body and the job within the simulated model complete. Note that it matters not which happens first in *real* time, nor does it matter whether the virtual time predicted by the model is greater or smaller than the real execution time of either the model or method. This is the key advantage of the virtual time execution framework. In Figure 8.1, which shows the execution progress in real time, the sample execution time for method M happens to be greater than the actual (real) time taken to execute M's body. Once the completion time, t_M, of M has been determined (dashed vertical line in Figure 8.1), any other threads whose virtual timestamp is less than t_M will be allowed to resume. T_1 itself will only be resumed when t_M becomes the smallest timestamp among the various threads, as shown in the diagram.

Once a method and its associated model job "join", the thread is assigned a virtual timestamp according to the performance model and waits to be rescheduled. This allows us to guarantee a correct virtual time schedule, ensuring that the thread T and the rest of the simulation is consistent with the model prediction.

The key idea of this approach is that jobs within a model are associated with threads on a one-to-one basis. This means that any background load will need to be modelled by one or more separate threads whose sole purpose will be to inject virtual jobs into the queueing network.

8.1.3 Guaranteeing functional consistency

In the trivial case where a model-simulated method is a stub, the thread T that enters it returns immediately and starts waiting for its associated job to "pass through" the model. A model-simulated method with a defined method body executes "outside" the VEX framework, because the actual (real) execution time of the method has no direct bearing on the VEX thread schedule, which is based on virtual time. By "outside" we mean that the methods executed by the associated threads are instrumented as usual (for example at method entry/exit and

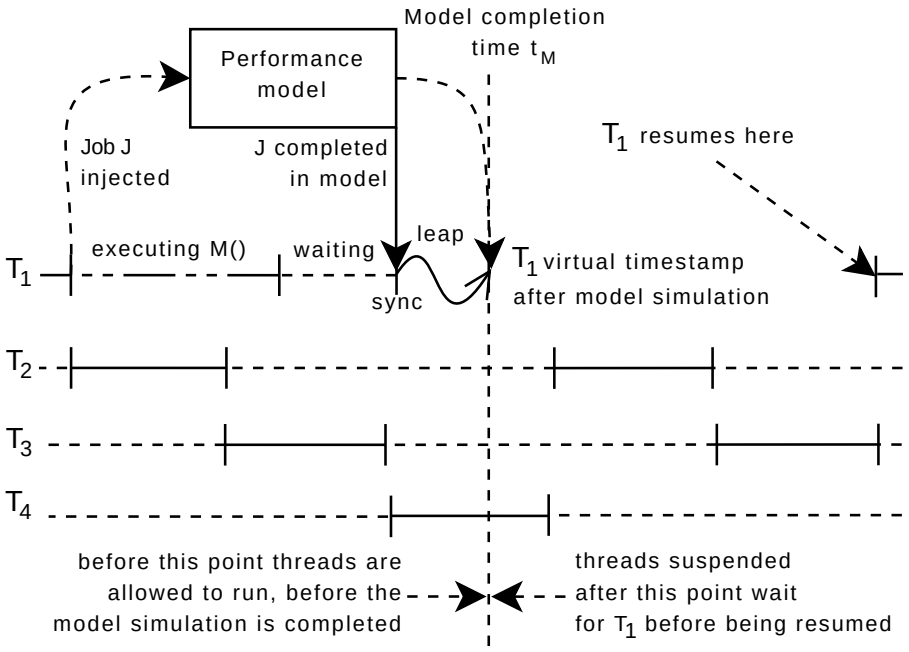


Figure 8.1: Simultaneous execution of methods body and performance model (shown in real time). Thread T_1 triggers the arrival of a new job in the model describing method M and continues to execute the body of M. Other simulation threads are scheduled according to the sample method execution time, as determined by the model.

synchronisation points) but that VEX events that are triggered at those points are ignored by VEX.

This simplifies the treatment of model-simulated methods but it can lead to a problem: what happens if the method blocks at a synchronisation point? There is now a subtle combination of events that can cause the VEX-controlled execution to deadlock, as illustrated in Figure 8.2. Suppose that the job has completed its course within a model and has produced a sample method completion time, i.e. point (1). Now assume that while the model is being simulated and while T_1 executes the code of method M, the VEX scheduler resumes another thread T_2 which enters a synchronised block on some object *obj* (2), before suspending when its timeslice expires (3). At this point, the model-predicted virtual timestamp of thread T_1 is the smallest in the simulator so the scheduler waits for the completion of M before resuming T_1. When T_1 tries to enter the monitor for *obj* it blocks (4), and the VEX-controlled execution deadlocks. We note here that it is perfectly possible for a thread to keep on executing a method for any amount of time, whilst the rest of the simulation is waiting on it to "join" a completed model-job. It is the lack of any progress that brings the system to a halt.

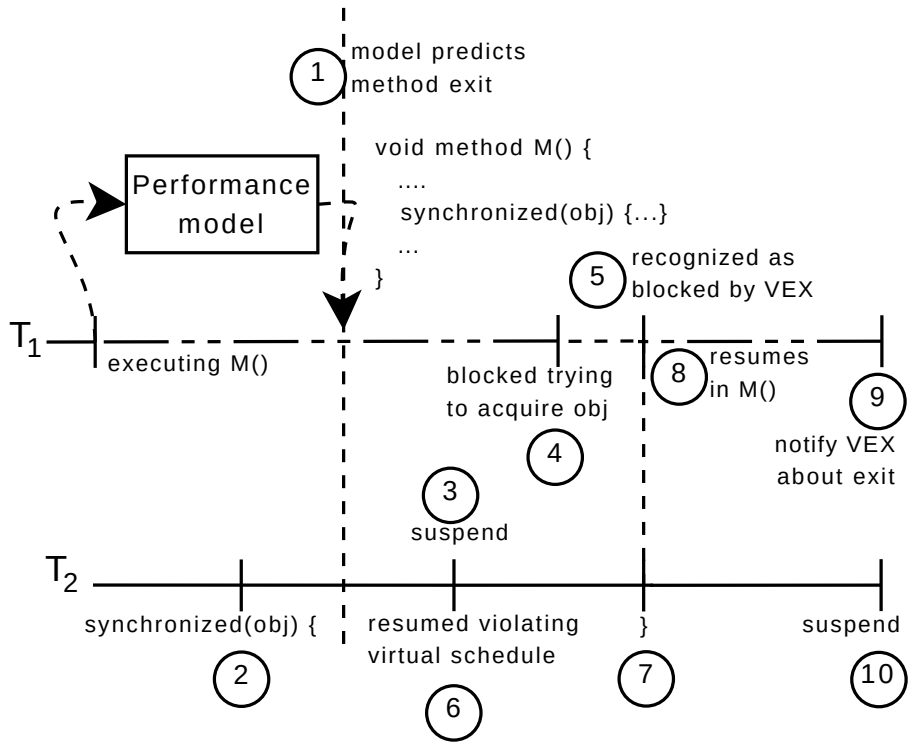


Figure 8.2: Description of the scenario, where the body of a model-simulated method leads the executing thread to block: the thread is recognised as blocking by the VEX scheduler and another thread is allowed to make progress.

In the prototype implementation of the model integration we solve the problem by implementing a deadlock detection algorithm. We set up a thread that periodically polls each thread that is executing the body of a model-simulated method to identify whether it is making progress or is currently blocked. If a thread is blocked (point (5) in Figure 8.2 for T_1, for example) and no other thread is allowed to execute before it in virtual time, then the strict virtual-time ordering is suspended and another thread T_2 is allowed to resume (6). If no other runnable thread exists, then the deadlock is an artefact of the application itself. Allowing other threads to make progress will eventually lead to a real code event (in this example release of the monitor of *obj* at point (7)) that resumes the blocked thread (8). This particular situation results in a temporary violation of VEX's normal scheduling policy, but such situations are rare in practice.

Note that an arguably more elegant solution would be to trap the event corresponding to the blocking of a model-simulated method within VEX, whilst ignoring all other events associated with the corresponding thread. However, in the case of Java, this depends on being able to detect synchronisation points within the JVM itself. The only way such events can be detected

by VEX is to modify the underlying JVM and that is something we wish to avoid, at least at the present time.

8.1.4 Merging resource contention

Performance models simulate contention for virtual resources, whilst real programs use real resources. We merge the two cases by treating performance models as virtual resources and monitoring the contention of the application's threads on them. The performance prediction of VEX is the result of the contention on both system and virtual resources. Threads either execute regular methods consuming real resources or contend for the model's virtual resources, after triggering the arrival of new jobs in a queueing network. Crucially, the state of the threads is monitored in both cases to distinguish between running, runnable, blocked or timed-waiting threads, and in order to schedule any other threads controlled by VEX.

Virtual resources are characterised as *local* or *remote*, depending on whether or not they contend with real code executing threads for the system hosts' CPU. This allows us to simulate CPU contention between real code and performance models that contain "local" resources. For real code execution the scheduler sleeps for time t to allow the executing thread to make one timeslice of progress. If the thread is associated with a job in a queueing network then the thread's virtual time (and VEX's global virtual time) is advanced by t. Remote resources cause the threads that "execute" them to block for an amount of virtual time specified by the queueing model. CPU resources remain available to any runnable (or local-resource simulating) threads.

8.2 Integrating open Queueing Networks with VEX

The objective is now to define the performance models that will be used in the virtual time execution to simulate the performance of methods. These performance models are open queueing networks, although there is nothing in principle that precludes the use of other types of

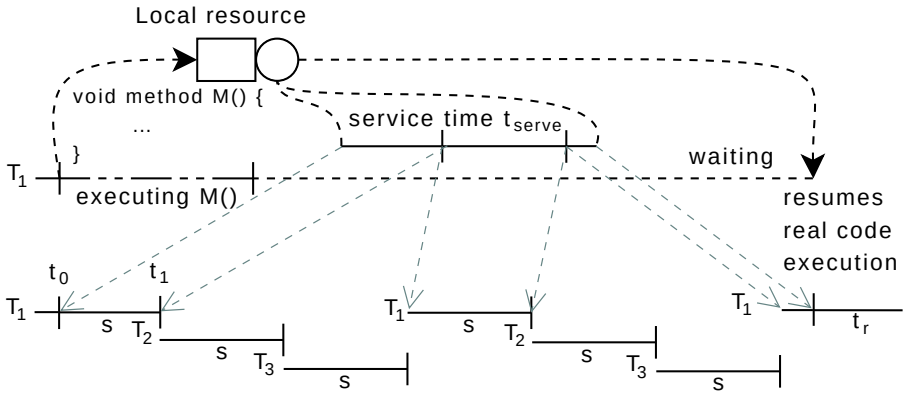


Figure 8.3: Local resource simulation: T_1 enters a local resource and its job is scheduled in a round-robin fashion with threads T_2 and T_3 that are executing real code. The scheduling of the job concerns subtracting a timeslice from the model-estimated service time.

models, such as stochastic Petri nets, stochastic process algebras or a bespoke discrete-event simulation. An advantage of queueing networks is that notions like threads "running" or being "blocked" are trivially matched to jobs being "serviced" or "queueing" respectively in the model. Similarly, method entry points correspond to *source* nodes and exit points to *sink* nodes.

As explained in Section 8.1.2, jobs are injected at the source nodes of a queueing network when a thread starts executing the method described by it. When a job reaches a sink node, it synchronises with the thread that corresponds to it (see 8.1.1). Routing decisions are determined by the model's parameters. We handle queueing by setting the state of the thread T_j that corresponds to the queueing job as blocked. Since T_j cannot make further progress until the resource is released, the VEX scheduler resumes a different thread. Note that only the VEX internal state for the thread is set to be blocked; the thread itself remains runnable as far as the operating system is concerned and is able to make progress executing the body of the method whose performance it is simulating. The same applies for other similar state changes described here.

An important feature is that existing performance tools, like workload generators, profilers and analysis tools can be used in conjunction with VEX. By replacing the system timestamps with corresponding virtual ones, VEX will automatically deliver virtual timing results, possibly

generated from a performance model, to any framework that invokes it. As an example, the JUnitPerf [1], that builds on the popular JUnit [73] unit-testing framework, can enable VEX simply by invoking the appropriate JVM parameters to turn a real time performance test into a virtual time one. The idea here is to set up a JUnitPerf test so that a test will fail if the total estimated *virtual* time is higher than a specified limit. We use this approach in the case study presented in Section 8.4.

8.2.1 Defining method-level performance models

The queueing networks we assume are generated by the open-source Java Modelling Tools (JMT) suite [16]. Specifically, we export the XML descriptions generated by the *JSIMgraph* application of the suite, that provides a user-friendly interface for defining the structure of open queueing networks. This includes the defined job classes, the connection between elements and the corresponding routing probabilities, the service-time distributions and the queueing policies of each queue. We modified the GUI of JMT to allow the characterisation of the virtual resource consumption of servicing nodes, as "local" or "remote" (see Section 8.1.4) and amended the related parameters to the exported XML file.

Queueing networks are assigned to methods via an extension to the annotation mechanism of the Java Instrumentation Environment (JINE) (Chapter 5), that interfaces Java applications to the VEX core, by instrumenting the bytecode of each class during class loading. Although the only required parameter is the name of the XML file that was exported by JMT, the annotation interface supports three optional parameters:

- Whether the body of the simulated-method should be replaced or not (via bytecode instrumentation).
- The label of the "source" node (as defined by the *JSIMgraph* application), to which new jobs are added, when this method is executed.
- The class of the jobs created from this method.

The overall effect of the last two parameters is that methods may contend for the same virtual resources in a different way, depending on the source nodes they are injected into and their associated job (customer) classes. As an example, the annotated method:

```
@virtualtime.ModelPerformance (
  jmtModelFilename  = "models/qn.jsimg",
  replaceMethodBody = false,
  customerClass     = "Class0",
  sourceNodeLabel   = "Sell_Transaction")
void M() { ... }
```

causes the body of the method M to be executed as is, but for its execution time to be determined by a queueing network model located at "models/qn.jsimg" by injecting a virtual job of the class "Class0" at the source node labeled "Sell_Transaction". If the source code is not available, or for testing reasons, the use of annotations can be replaced by a file mapping method Fully Qualified Names (FQNs) to model-simulation parameters.

The XML file names that describe the queueing networks for model-simulated methods together with the optional "source" node and customer class parameters, are registered to the native-level VEX core, which uses a low-level XML parser to extract node information. In turn, this generates a queueing network object which is processed by a C++ implementation of the JINQS Java simulation framework for multiclass queueing networks ([45]); we refer to it here as "CINQS" for want of a better name. We note that the CINQS core, including event scheduling and the management of virtual simulated time, simply defers to VEX, which already supports this functionality.

8.2.2 Assumptions

The prototype implementation does not support all of the features of either JMT or JINQS so the queueing networks have some limitations at present. In particular, we currently exclude

finite-capacity regions and class-dependent routing and assume that jobs that are rejected when a finite-capacity queue is full are forwarded to the sink node of the corresponding queueing network. None of these restrictions are significant and all of them could be straightforwardly relaxed with additional work.

VEX is, however, currently tightly-coupled with the queueing network approach. Although the CINQS module is separate from the core of the VEX framework, an integration driver is used to convey events in the queueing network to the scheduler. Nevertheless, the idea of integrating performance models and real code is in principle applicable to other types of models. One could consider replacing the simulation of a model by a performance simulator, which would have to be synchronised with code prototypes in virtual time (much like the SliceTime approach in [143], for example). This would be perfectly possible as long as a specific "driver" were used to interface VEX with the custom simulator. We raise these issues again in Section 9.2.3.

8.2.3 Simulating models in VEX

Suppose that a thread T with a VEX-maintained virtual timestamp t_0 enters a model-simulated method M. The method entry traps T into the VEX core and a new job J is added to the queueing network associated with M. At this point the operating system thread associated with T has the (sole) responsibility for executing the body of M (i.e. *outside* the VEX framework) whilst the progress of the virtual job through the queueing network is administered by the VEX scheduler. The thread T next comes under the control of VEX when it completes M and synchronises with J on its departure from the queueing network – see Section 8.1.2. Before this synchronisation happens, the operating system thread associated with T is essentially "invisible" to VEX but, importantly, VEX will update the virtual timestamp it associates with T as J progresses through the queueing network. It is important to understand the distinction between the operating system thread associated with T and the additional information about T that is maintained internally by VEX. Remote and local nodes within a queueing network are thus handled as follows:

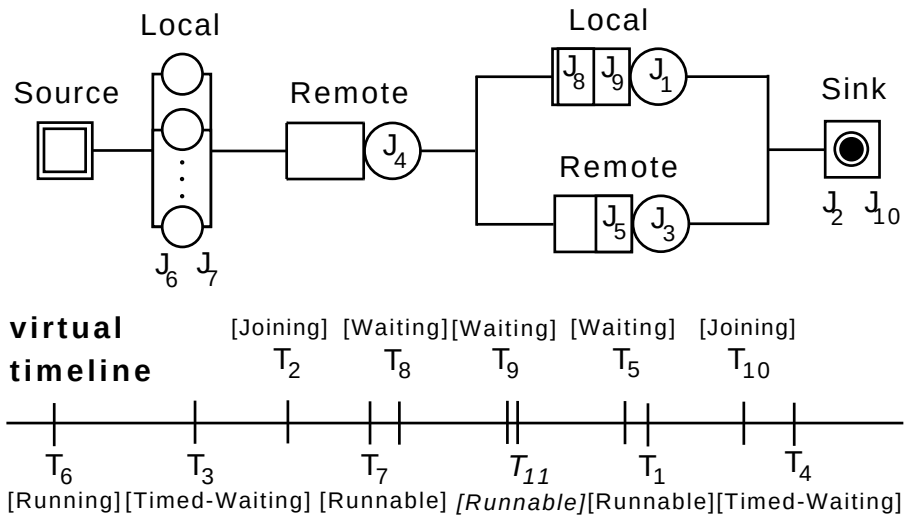


Figure 8.4: Matching of VEX thread-states to the states of their jobs in a sample queueing model

- If J goes into service at a "remote" server (see Section 8.1.4), then VEX assigns T a new virtual timestamp $t_1 = t_0 + t_{serv}$ where t_{serv} is the sample service time of the server. Once t_1 becomes the smallest virtual timestamp amongst the threads managed within VEX, the scheduler removes J from the server and moves it onto the next node in the queueing network. VEX thus increments the (internal) virtual timestamp associated with T, mirroring the progression of J through the queueing network.

- If J is serviced in a "local" queueing node (see Figure 8.3), then the residual service time, r say, is given by $s - t_{serv}$, where t_{serv} is the service time. If $r > 0$ then the virtual service completion will have happened within the current timeslice so J can progress to the next node. If $r \leq 0$ then $-r$ represents that part of the service time that must happen in the next timeslice. We thus assign $t_{serv} = -r$ and give T a new virtual timestamp $t_1 = t_0 - r$ *within VEX* and add T to the list of runnable threads being maintained by VEX. A virtual job can thus pass through several nodes in a queueing network during a timeslice.

At any point in time a simulated queueing network can thus contain many virtual jobs, each of which is associated with a (real) thread in the application. If a job starts queueing at a node (remote or local), then its associated thread state changes to "Waiting", denoting that the thread should not be scheduled to progress until its state changes. To illustrate this, Figure 8.4

shows an example of a queueing network with ten jobs (application threads) located at different queues/servers in the network. Job J_i corresponds to thread T_i, whose virtual timestamp is depicted by its location on the virtual timeline and whose current state is maintained internally by VEX. Threads illustrated below the timeline are resumed when their virtual timestamp becomes the smallest in the simulation, while threads shown above the timeline are waiting for an event to happen before they can be resumed. There are two possible events: the releasing of a resource in the queueing network and the completion of a job, which in turn represents the completion of a method in virtual time. Thread T_{11} corresponds to a thread executing real code. Although it is not associated with any job in the queueing network, it is treated similarly to any other thread.

In the context of a single CPU execution a key difference between the way jobs and real threads are handled is that real threads are always resumed at the virtual time point at which the last running thread changed its state (becoming runnable, waiting etc). The same approach is also followed by jobs that are serviced by a local resource in a queueing network. This is shown in Figure 8.5 (lower part). In contrast, jobs that are being serviced by remote resources can resume the model simulation at the virtual timestamp defined by the previous remote resource, regardless of the current global virtual timestamp, as illustrated in the upper part of Figure 8.5. In this way, a model simulating a series of remote resources may run in parallel with any locally serviced or real-time executed threads.

VEX synchronises the event corresponding to thread T completing the execution of method M with the associated job exiting the queueing network. At that point the virtual timestamp of T, t_x say, is the one determined from the queueing network and the VEX-maintained CPU time of thread T is updated accordingly. The estimated duration of M is $t_x - t_0$.

8.3 Application in performance engineering

In this section we summarise our vision on how the model-integrating version of VEX can support performance engineering throughout the software lifecycle. Although this approach is

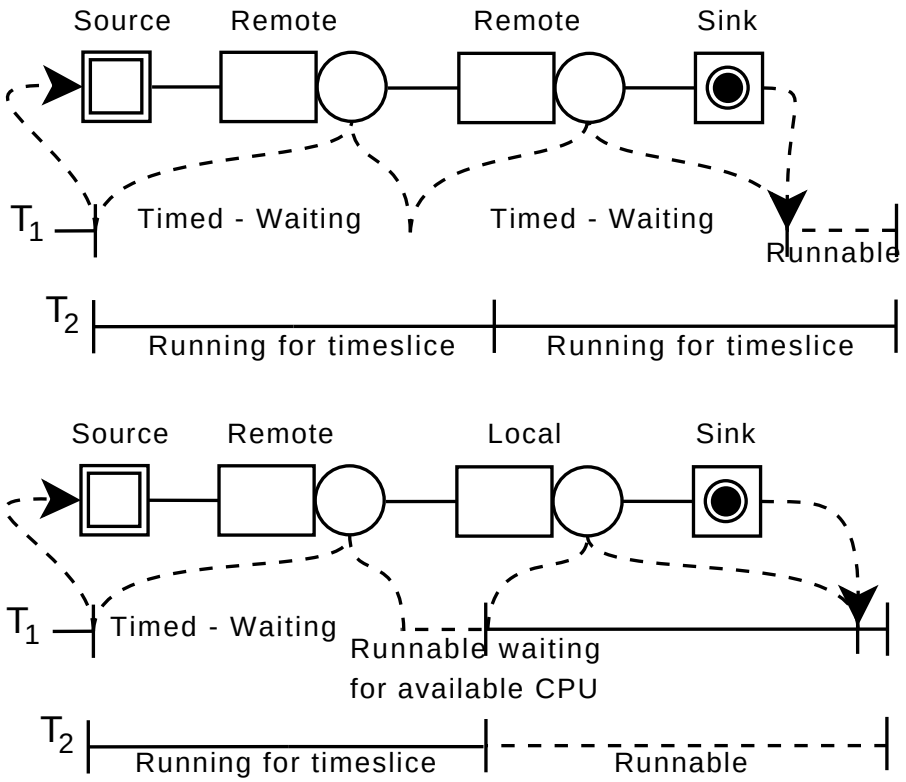


Figure 8.5: Differences in virtual time scheduling after a remote resource, depending on whether the next node belongs to a local or a remote resource.

applicable to any software engineering methodology, it fits very well with existing processes from test-driven agile development. Specifically, in the beginning of every iteration, a number of use cases (or "stories") are defined. Tests for these scenarios are implemented and a code skeleton is set up to enable functional testing. This structure comprises code stubs, which are methods that return a value, or mocks, that simulate the behaviour of objects. The idea is that the performance engineer then identifies the scenarios with the highest performance risk and develops performance tests to compare their execution times with the predefined performance targets for each scenario. The performance tests execute/simulate the code of the corresponding use-case in virtual time through VEX. This means that the predefined performance targets are compared with virtual execution times, i.e. accumulated virtual times on the timeline of VEX, rather than real execution times.

For Java applications the stubs and mocks of the performance critical path are annotated with the XML files and parameters that correspond to open queueing networks that describe their performance. The parameters of the network nodes correspond to service times of different

resources, such as critical sections, external servers, CPUs of the simulating system or any other shared system resource. Job arrival rates of the models are not taken into account. These parameters can be populated as best- and worst case estimations, as in SPE [125] and two corresponding performance tests can be defined for every scenario. Failure of the best-case scenario performance test is considered as more critical than the worst-case one.

Having setup the stubs, the tests and the models, the idea is for the development team to write code with a view to passing all tests successfully - both functional and performance ones. Similarly to what happens when functional tests fail, performance test failures are notified to the performance engineer. VEX provides a profile of the integrated application and supports "what-if" performance prediction via update of the model parameters. When all performance and functional tests are successfully passed, the performance engineer repeats the performance critical tests on the complete code base in *real* time without VEX. The real time performance tests *should* be successful, though to an extent this depends on the selection and parameterisation of the models. This last step constitutes model validation.

Consider now what happens when a new "story" is to be incorporated in the code, as it often happens in agile development. The use-case is implemented by a new method N, which requires access to the same sources of an existing method M. Virtual time execution provides performance predictions immediately, by re-using the model that describes M and enhancing it with the features of the new method N. The engineers either annotate the code of M with the new model, or define a subclass and override M there, associating the subclass's method with the new model. Any other method is "modelled" in virtual time using its existing code. When N is implemented, the annotation of M is removed, and the complete code base is performance-tested once again.

We note here that frequent application-wide testing, as the one performed during continuous integration, may require a significant amount of time. In contrast, the duration of performance tests in virtual time can be extremely fast, especially when many parts are simulated as models.

8.4 Case study

We demonstrate how models are integrated in VEX by walking through the development of a contrived server-client application based on a MySQL server. The purpose of this section is to demonstrate the ability of VEX to allow models and code to 'cooperate' in a performance engineering exercise and to deliver predictions that are 'correct', in that they correspond to whatever models happen to be provided.

It is important to appreciate that this is *not* meant to be a performance evaluation case study. Nor do we attempt to devise accurate or 'realistic' performance models for the various parts of the case study application. Indeed, we have intentionally devised very simple models in order to focus on the tools and methodology, rather than the case study itself. For the purposes of this exercise the accuracy of the models we refer is therefore largely immaterial.

The exercise mimics the VEX-based performance engineering process that we envisage during the development of an application. A key aspect is that at every point we have some combination of code and performance models that can be used to facilitate performance testing:

- At the outset of a project we specify quantitative performance targets for the application, which might typically define minimum acceptable Quality of Service requirements or Service-Level Agreements.

- The first phases of application development begin with a pure performance model, or possibly a skeleton implementation that is capable of making calls to, as yet unimplemented, methods corresponding to the key top-level components of the application. Any unimplemented code sections are stubs that have associated performance models and may compute some minimal synthetic result so that they function correctly upon their invocation.

- As the development proceeds, we gradually replace stubs with code which, in general, will render the associated performance models obsolete. As the application development proceeds, so the performance model predictions are replaced automatically with results

from code profiling.

- The process ends with a complete application which, we hope, meets its specified performance targets. As in the Software Performance Engineering methodology proposed in [125] we can validate the models used throughout the process against the final code. We essentially get validation for free.

- At any point the performance implications of any alternative implementation or design decision can be explored by replacing measured method execution times with predictions from a model or by time-scaling observed measurements.

VEX thus integrates traditional model-driven and profile-driven performance analysis into a single framework.

In the scenario we consider the client-side issues a number of requests R to a server at a specified average rate λ. Each thread will first check a local cache to see whether its request is served locally. If not, the request will be sent to, and serviced by, a remote MySQL server that has its query cache disabled. The result returned by the server will be cached for future use; the local cache is assumed to use an LRU replacement policy. There are N different request types which are issued as SQL queries to the server, which accesses the database of the TPC-W benchmark [49]. The server always returns an integer.

The cache comprises a buffer of size S_c and is direct-mapped using a key that corresponds to the request type. The maximum number of request types is N so, assuming the cache elements are accessed randomly, we can prescribe the size S_c required to generate a hit rate of $\rho = S_c/N$. ρ is used in a queueing model of the system to determine the routing probability from the request source to the MySQL server; in the code it is used to determine the cache size ($S_c = \rho N$).

The various parameter values/distributions we assume are summarised in Table 8.1. All measurements have been taken on Host-2. In this example scenario we only consider single CPU hosts and so the second core of the system is disabled. A number of dummy requests is issued prior to the actual measurement start to limit warm-up effects from the JVM. For the MySQL server we use version 5.5.

No. of requests, R	3000
Request arrival rate, λ	400
No. of request types, N	52
Cache size, S_c	39
MySQL service time distribution	$Det(0.007)$
Cache service time distribution	$Exp(20000)$
Cache hit rate, ρ	0.75

Table 8.1: Parameter values and distributions used in the modelling case study

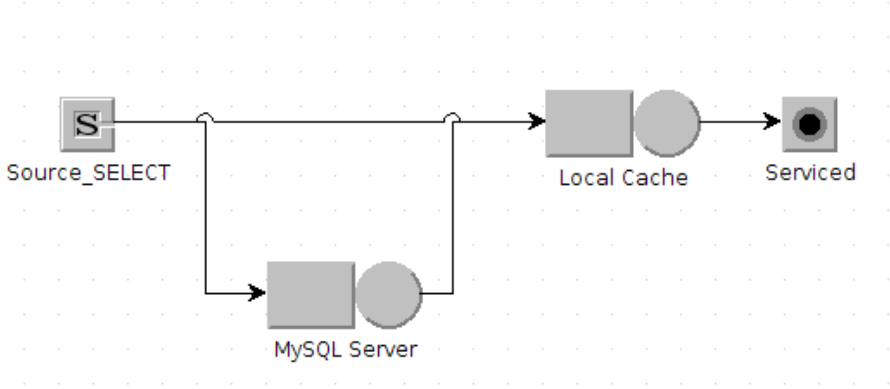


Figure 8.6: demo_select.jsimg: Queueing network for the Pure model version

8.4.1 Pure model

We start the process by defining a code skeleton for the client, which spawns a new thread for every new request. It then "thinks" for an amount of time by calling the *think()* method of a ThinkingBehaviour interface. Initially, this method is described by a single infinite server node sampling from an exponential distribution with rate λ. This simulates the time that the client spawning thread is sleeping without occupying a resource. Each spawned client then invokes the service by calling the *service()* method of a ServiceBehaviour interface. This method is originally simulated by the model of Figure 8.6. In this model, the "Local Cache" is simulated by a local infinite server node and the "MySQL Server" by a remote single-server queueing node. Consistent with the assumptions set for the case study, the routing probabilities from the source to the two nodes are ρ for the "Local Cache" and $1 - \rho$ for the "MySQL server". Running a virtual time performance test on the version with the model-simulated *think()* and *service()* methods for R requests, we acquire the results of the "Pure model" line in Table 8.2.

```
@virtualtime.ModelPerformance(
        jmtModelFilename="/homes/nb605/VTF/src/models/demo_select.jsimg"
        replaceMethodBody=true)
public void service(Request request) {
    Assert.fail("This method body should be replaced during runtime");
}
```

Figure 8.7: Code for unimplemented *select()* method.

8.4.2 Client-thinking implemented

In the next step we implement the thinking behaviour by executing real code that invokes *Thread.sleep()* instead of simulating an infinite server. The sleep duration is defined by the same exponential distribution as the model (with rate λ), while the rest of the unimplemented code remains the same. We repeat the performance test with VEX and acquire the results of the second row of a Table 8.2. The expected total time of the method remains close to the "Pure model" one, though the standard deviation is higher, as measurements on (the now-larger) implemented code increase the non-determinism of the results. We note here that if instead of sleeping, a thread performed some work, it would contend for the same resources as the "Local Cache" node of the model of Figure 8.6.

8.4.3 Partially-complete code

We start implementing the *service()* method by developing the cache handling. The new form of the *service()* method is shown in Figure 8.8. The model-described method is now the *makeRequestToDBserver()* method, invoked only upon a cache miss. The body of this method is actually being executed returning always the same value. Its performance behaviour is defined by the model provided in its annotation, which is depicted in Figure 8.9. This shows clearly how real code and performance models are integrated in VEX. The control flow determined by the execution of the real code determines the job arrival process in the performance model. For the purposes of this exercise we have used a prescribed cache hit rate, whereas it would be equally possible for the workload to be driven by a more elaborate caching scheme that would not be as easy to model in a queueing network. Performance testing this version of the code we get a similar execution time, as shown in Table 8.2.

```
@virtualtime.ModelPerformance(
        jmtModelFilename="/homes/nb605/VTF/src/models/demo_select_only_d
        replaceMethodBody=false)
protected Object makeRequestToDBserver(Request request) {
    return 53;
}

public void service(Request request) {
    int result = 0;
    int key = (int)request.getChar();
    Object cachedValue = cache.retrieve(key);
    if (cachedValue == null) {
        Object value = makeRequestToDBserver(request);
        cache.add(key, value);
        result = Integer.valueOf(value.toString());
    } else {
        result = Integer.valueOf(cachedValue.toString());
    }
    request.setResult(result);

}
```

Figure 8.8: Code for *select()* method with implemented cache. Note that the value returned by the DB server is arbitrary (here 53 as shown).

Development stage	**Time [ms]**	**Stdev [ms]**	**Error (%)**
Pure model	7857.10	53.366	-16.53
Client-thinking implemented	7751.67	145.938	-17.65
Partially-complete code	7800.30	158.632	-17.13
VEX complete code	8576.93	123.045	-8.88
Real complete code	9413.07	140.633	0

Table 8.2: Total times of performance tests in the different development stages of case study using the modelling integration feature of VEX (30 runs per result)

8.4.4 Complete code

Completing the code base, we replace the body of the *makeRequestToDbServer()* method (code in Figure 8.10) to issue the request to the MySQL server running on Host-3 from Appendix A on the same local area network (code in Figure 8.11). Executing the performance test in virtual time results in a slight increase in the expected times (fourth row of Table 8.2). We regard this to be related to the overheads from real time I/O measurements, that also include some background system noise, due to the fact that the OS scheduler determines when the thread that is performing the measurement is resumed. Since the entire code base is complete, we can also perform the performance tests in real time. Doing so we acquire the result next to the "Real time" row of (Table 8.2), which is higher than the originally predicted values by approximately 17%. This is because the OS scheduler does not resume sleeping threads exactly at the time determined by each think-time distribution sample. Indeed, we measured a 12%

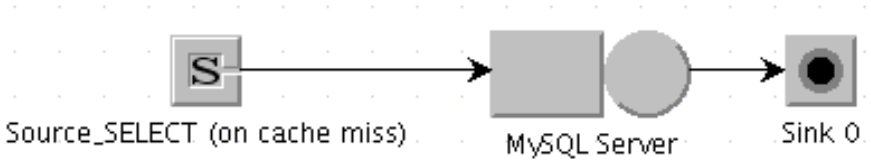


Figure 8.9: demo_select_only_db.jsimg: Queueing network for remote server only

```
protected Object makeRequestToDBserver(Request request) {
    try {
        Integer returnValue = info.getMySqlServerResponse(request.getChar());
        return returnValue;
    } catch (SQLException e) {
        System.out.println("SQLException on select " + e.getMessage());
        return 0;
    }
}
```

Figure 8.10: Code for the implemented *select()* method invoking method *getMySqlServerResponse* of class *SqlDriverInfo*

increase in the average think time of the real-time execution compared to the requested λ rate. This increases the total running time and decreases the load on the remote MySQL server, which also leads to lower response times as shown by the histograms of Figure 8.12. Although we could build this delayed resumption into VEX very straightforwardly, we have chosen not to tie the tool to a particular operating system or JVM. With that said, it would be a simple task to provide parameterisations of the tool for individual platforms with such known 'artefacts'.

8.4.5 Cache study

To demonstrate a simple validation test, we might wish to compare the effect of the coded cache size S_c of the "Partially-complete code" section, to the one observed in real time. The total test time per cache size $S_c = 1,\ldots,N$ for the "Real" and the "Partially-complete code" are illustrated in Figure 8.13. This presents a good prediction accuracy for VEX for lower cache sizes S_c. The prediction error is increased for higher values for S_c, due to the differences in the think-times between the real-time execution and those in the partially-complete code simulation (see Section 8.4.4 above).

```
public Integer getMySqlServerResponse(char c) throws SQLException  {
    PreparedStatement getCount = cachedConnectionToMySqlServer.
        prepareStatement(getQueryStringOfAllRecordsWithLetter(c));
    ResultSet countResult = getCount.executeQuery();
    countResult.next();
    Integer count = countResult.getInt(1);
    countResult.close();
    return count;
}
```

Figure 8.11: Implementation of *SqlDriverInfo.getMySqlServerResponse* that makes the request to the MySQL server.

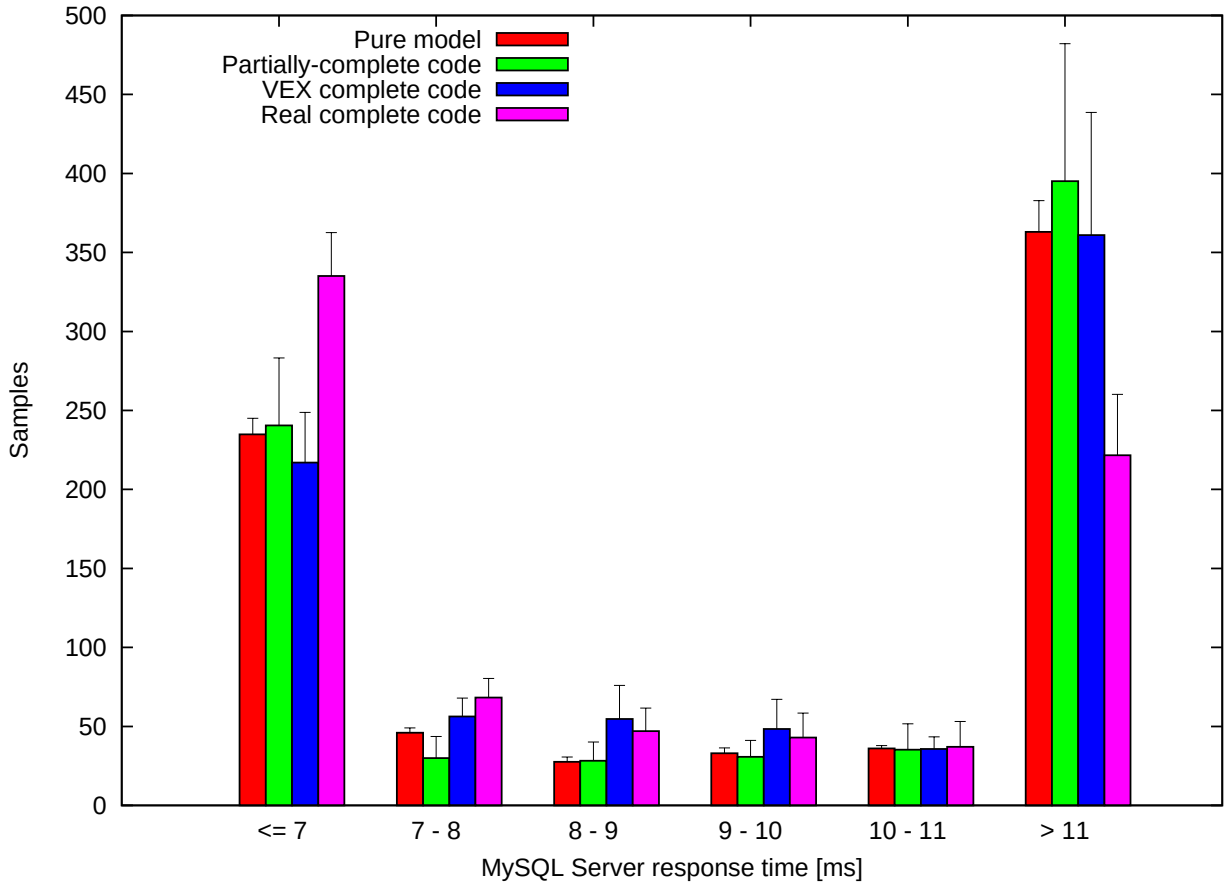


Figure 8.12: Histograms of MySQL Server response times for different development stages with 95% confidence intervals (10 runs) showing consistent results for VEX simulations and lower response times for real executions.

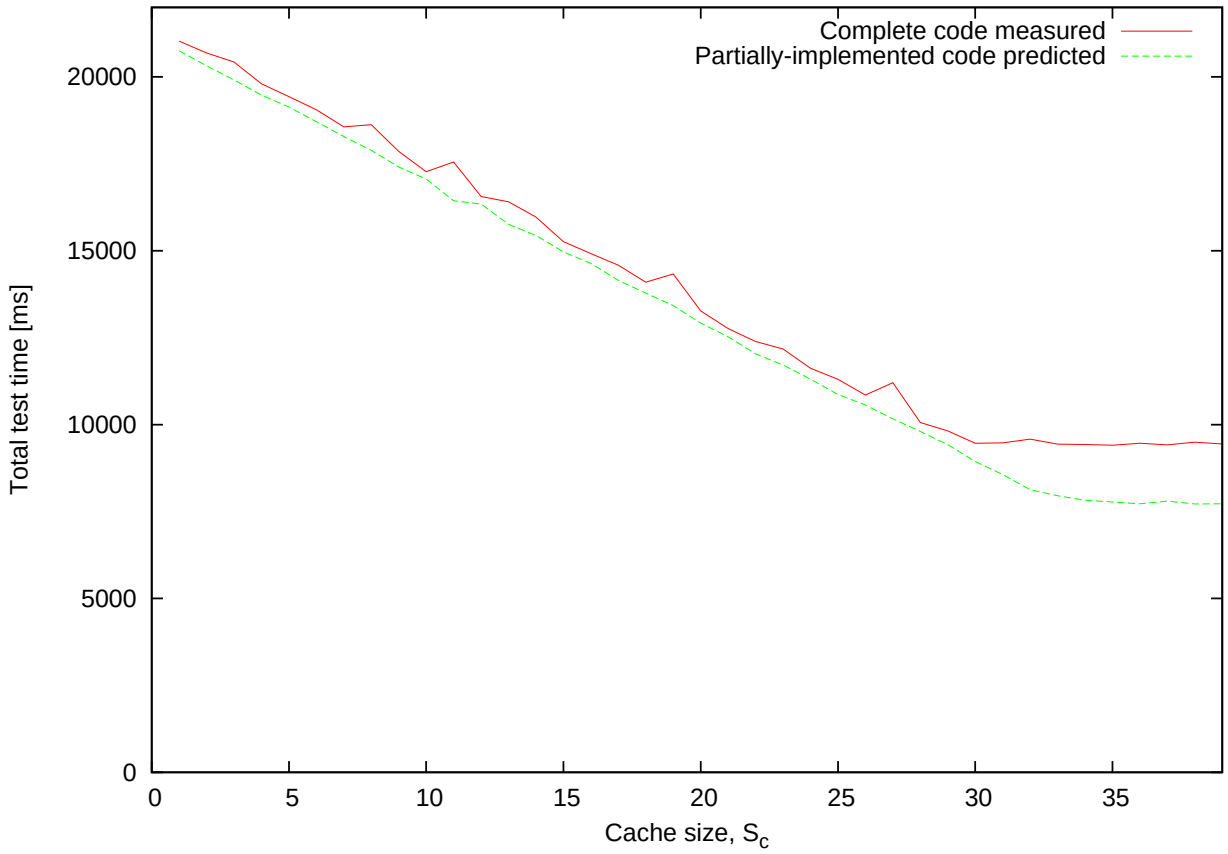


Figure 8.13: Effect of cache size in VEX simulation of partially-complete code and complete-program execution. The mean of 10 runs per S_c is shown.

Chapter 9

Conclusion

9.1 Summary of Thesis Achievements

In this thesis a new approach for software performance engineering has been presented. We have introduced the notion of virtual time execution, as a language-independent high-level execution-driven simulation [32] approach, that offers performance predictions for hypothetical performance scenarios at the method-level. Time is either modelled as a linear transformation of observed real time or as an arbitrary transformation determined by the simulation of a performance model. By doing so, we are able to simulate the execution of an application without requiring any low-level system models. We have explored the tradeoffs between simulation overhead and performance prediction accuracy in this context and have observed that our approach provides performance predictions that are typically within 10% of the observed real time at overheads of less than 1.5. However, fully profiled codebases and some short running applications and stress tests led to significant outliers.

We have designed and developed a prototype for our methodology, by combining a language-independent simulation core (VEX), with a Java-oriented instrumentation framework. During the verification and validation process on synthetic and real benchmarks, we investigated the impact of non-trivial effects to the virtual time methodology. We explored how the execution of the framework's instruments, scheduling decisions or JVM features like JIT compilation modify

and affect the profiling of the application code.

Our approach differs from previous work on instrumentation intrusion [87, 86] and compensation [37, 42] in that it has to deal with the fact that the framework itself controls the execution progress of the code it is profiling. We discussed how other JVM features like safepoint synchronisation and garbage collection modify the states of threads at a lower level than VEX can monitor, leading threads to run uncontrolled (assumed as waiting at a native synchronisation point) or wrongly perform time warps in virtual time (virtual leap forward). We presented extensions to the VEX code to cover such cases, via adaptive profiling, the management of native waiting threads or heuristics, and identified how "uncaught" effects are typically responsible for the high simulation mispredictions. Thus we highlighted the limitations of the language-independent core for Java applications and of execution-driven simulation in general at this high level.

A separate component of our work concerned the simulation of I/O operations in VEX. To account for such operations correctly, and simulate DMA in virtual time, we developed an empirical runtime I/O-duration prediction scheme [50, 151, 122] and have investigated the accuracy of various black-box prediction methods for the entire I/O subsystem. Each I/O invocation point is identified via a unique id that is transparently added using dynamic bytecode instrumentation techniques [23] on the application classes and static instrumentation on the system classes. We have proposed a solution for time-scaling and a solution for random exclusion of I/O operations as means to identify the overall performance effect of I/O on the application. A threshold-based method to distinguish cached predictions has also been introduced. Our methods have been evaluated on custom disk I/O stress tests and JDBC database drivers, providing low prediction errors, showing how underpredictions have a significantly adverse effect on prediction (in contrast to overpredictions). We also show that using an I/O threshold, based on a histogram of I/O measured samples, improves the simulation accuracy.

In order to model a multicore simulation host, we have addressed the issue of parallel virtual time execution, by fitting well known conservative time window [82] synchronisation schemes to VEX. One such approach allows parallel progress until an action that decouples virtual time

from real time progress occurs. The other approach uses a quantum-based synchronisation scheme for all cores, that uses the VEX scheduler timeslice as a barrier. These methods have been compared for our benchmarks generating predictions of similar accuracy as the uniprocessor simulation, when the number of virtual and physical cores match. We also demonstrate the framework's ability to simulation more cores than the ones physically available on the simulation host, although this is not its intended use.

Finally, we outlined a vision for continuous performance testing in virtual time, by simulating the durations of unimplemented, or even completed, methods with performance models. Queueing networks have been used as examples to describe local or remote resources that respectively contend or not for access to the CPU with threads that execute code. Code execution has been decoupled from its timing aspect and a synchronisation scheme, that takes into account JVM effects like native waiting, has been proposed to match the functional and timing aspects of the virtual time execution. We have also demonstrated, by way of a case study how our methodology might be used to performance test a database client application.

Overall our results demonstrate that, despite the various language-specific issues we encountered, the virtual time execution idea presents significant potential for future research to offer a novel lightweight technique for performance testing throughout the software lifecycle.

9.2 Open Questions

9.2.1 Performance prediction quality

The performance predictions provided by VEX have presented considerable variability. Although predictions are generally within a 10% error, there have been cases that due to observer effects and changes in JIT compilation schemes, major page-faults or program behaviour (native waiting states or safepoint synchronisations), for which the prediction error was considerably higher. This applies for both single- and multi-core architectures.

To deal with this issue we could quantify the expected accuracy of our approach. For instance,

we could provide the user of the framework with warning signals that would indicate a risk of mispredictions in the results reported by VEX. This is accomplished by measuring simulator events that we have found to have a negative impact on prediction quality and define thresholds over which we would issue warning signals.

For example, observer effects could be quantified by the total number of invocations to VEX instruments, including method calls, I/O functions, synchronisation routines etc. The higher the rate of such traps to the execution time of the simulation, the higher the expected prediction error. The total number of major page-faults, whose duration is not accounted for in VEX, is monitored by system counters. The frequency of transition to the native waiting state or the number of invocations to *Thread.yield* are also quantifiable and can be used as indicative of the expected prediction quality.

We have not validated the time-scaled results, except for cases where the expected results were theoretically known in advance. Excluding the implementation of an optimised version of a time-scaled code and timing it, using some kind of lower-level simulation to acquire the "correct" value for a time-scaling factor would be possible in principle. This would either require us to integrate a full-system simulator [84, 109] in VEX, convey to it the system state at the point of method time-scaling and scale accordingly the returned timing result or execute our benchmarks using a full-system simulator and modifying the timing results at the method level.

9.2.2 Evaluation on large scale applications

VEX and JINE have been evaluated in terms of short synthetic and well-known small and medium size multi-threaded client-side Java benchmarks, in an attempt to validate the simulation approach on various workloads. However, VEX has not been used in an industrial-strength scenario, that would allow it to address an existing performance problem. Indeed, the various component parts of VEX (single/multicore simulation, JINE, I/O simulation and models) have been presented more or less in isolation from each other - a larger case study that would involve all of them, would better frame our work and indicate any further issues from the interactions

of the components. The lack of such a study is attributed to the problems that have arisen in the simpler case scenarios due to JVM specific issues and to the fact that most large scale scenarios are multi-tier or distributed applications, that require a mechanism of synchronising different processes in virtual time. Our work on multicore VEX constitutes a first step in this direction.

9.2.3 I/O modelling

The I/O prediction scheme presented attempts to provide a high-level approximate empirical prediction for I/O operations. Although our evaluation studies converge to the conclusion that prediction schemes that favour overprediction yield good prediction results, there is no clear optimal prediction method. In this respect, we could apply an adaptive prediction scheme, similar to [151], that would evaluate different policies at runtime, before selecting the one that provides the most accurate I/O predictions. The same could apply for parameters like the size of the Markov chain, if that prediction approach was used, or the size of the prediction buffer. Our scheme could also be supported by a more accurate I/O threshold, that would be extracted from the simulation host, rather than being arbitrarily set by the user. However, only integration with the OS would definitely distinguish between cached and disk hitting I/O operations, thus alleviating the need for such a threshold.

Simulating a performance model (i.e. a queueing network), instead of measuring I/O durations might also improve the prediction accuracy of the online I/O prediction system. All traffic to the I/O subsystem (at the system call level) would be thus treated as job arrival events to the designated performance models. Developing a generic VEX-modelling-driver interface, we could even include a disk simulator [2] and map its predicted times on the VEX virtual timeline. Alternatively, we could use the observed I/O durations to calibrate models of the file system [101] and then simulate these models in the current or future virtual time executions.

9.2.4 Full integration in the JVM source

Another direction for future research is to integrate all functionalities of VEX and JINE within the JVM source. This would limit the discrepancy between the bytecode executed in real-time and instrumented bytecode of the virtual time execution and relax the requirement to preload a set of statically instrumented system classes. Conditionals or virtual methods in JVM classes could be used to distinguish between regular and virtual time executions. These effects could improve the prediction accuracy of the simulation, though it is not clear how significant this improvement might be. Moreover, still experimental features like the *explicit monitor trapping* and the *low-level* trapping to VEX within the JVM need to be properly tested and evaluated.

9.2.5 Optimisation choice selection

We have presented VEX as a simulator for analysing the sensitivity of the application's performance on a defined post-development optimisation choice, manifested as a time-scaling factor of a method. However, no attempt has been made to automatically identify the main performance bottlenecks and investigate how various time-scaling factors affect them. In turn, and taking into account the typically low simulation overhead of VEX, we can imagine various optimisation choices being compared against each other at runtime. Virtual performance metrics like response times, resource utilisation and thread waiting times could then be gathered for each time-scaling factor to form an optimisation problem to find sets of method time-scaling factors that yield the best performance improvements. These sets could then guide the performance optimisation process. Performance models could also be used in a similar way.

9.3 Future Work

9.3.1 Lower-level virtual time executions

An interesting future direction is to explore modifications to the underlying OS, e.g. by making scheduling decisions on the basis of virtual time. Special-purpose traps to VEX would have to be added to the kernel, to identify major page faults or any I/O operations hitting the cache. We would also be able to distinguish the device that each I/O operation targets, which allows us to investigate the effect of aggregating measurements at the I/O-device level to the accuracy of the I/O-prediction scheme. In principle, processes would be distinguished to the ones been scheduled in virtual time and ones running normally. The research questions here involve how this might be implemented, what would be the side-effects to the system operation and the effect this has on the prediction accuracy and overhead.

We have already discussed in Section 5.2.4 how using a lower-level integration of VEX and Java can convey thread state changes within the JVM to VEX. This approach is particularly helpful for the identification of "Native-waiting" threads, which remain otherwise "uncontrolled" by the VEX scheduler until they invoke one of our framework's instruments. This can lead to inaccuracies on virtual time accounting and virtual time scheduling, whilst complicating the management of virtual leaps forward. The custom JVM would follow the update-before-event principle, according to which upcoming state changes are conveyed to VEX, before they take place in real time. Besides the implementation complexity, the main disadvantage of this approach is that it would require that users execute their programs on the modified version of the JVM.

Preliminary results show that integrating VEX with the JVM indeed reduces the overhead of virtual time simulation. This comprises encouraging evidence that more radical changes to the JVM or the OS might well pay off.

9.3.2 Native code integration

Integration of VEX with a native binary code is expected to overcome many of the issues highlighted in this thesis, like safepoint synchronisation and hidden thread state changes, that have rendered the JVM simulation such a challenging task. On the other hand, we would have to find new and efficient ways to support features like adaptive profiling and class reloading.

9.3.3 Distributed VEX

Allowing different processes to synchronise on the same virtual timeline, would create a distributed VEX. This could follow the principles of Lax multicore synchronisation presented in Chapter 7, under which only specific events are conveyed from one process to another. Alternatively, we could identify and monitor the inter-process communication, correcting and synchronising only when messages are sent from one process to another (like PDES approaches [47]). At this point it is not clear, whether the Lax or this message-based approach would perform best in light of time-scaling and virtual time leaps. Developing the distributed VEX would enable us to apply our methodology to contemporary applications and address problems of significant performance interest.

9.3.4 Integration of other modelling techniques

Following the classification presented in [8], the behavioural modelling of VEX is based on real code execution (no-modelling) and the performance modelling of unimplemented methods in queueing networks. A possible extension could replace queueing networks with annotated UML profiles, that would then be transformed to the corresponding queueing networks. The objective here is to hide the mathematical modelling side and bring the procedure closer to the current practices of software engineering.

Appendix A

Experimental machines

A.1 Table of machines

Table A.1 presents the architectural details of the hosts that have been used throughout our experiments.

Name	Host-1	Host-2	Host-3	Host-4
Machine	nb605-laptop	scaramanga	irmabunt	nb605-laptop
Type	Core 2 Duo Mobile	Core 2 Duo	Core 2 Duo	Core 2 Duo Mobile
Model number	P8600	E6750	E6850	P8600
Cores	2	2	2	2
Threads	2	2	2	2
Frequency (GHz)	2.4	2.66	3	2.4
L1 instruction cache (KB)	2 x 32	2 x 32	2 x 32	2 x 32
L1 data cache (KB)	2 x 32 (write-back)	2 x 32 (write-back)	2 x 32 (write-back)	2 x 32 (write-back)
L2 cache (KB)	3072 (shared)	3072 (shared)	3072 (shared)	3072 (shared)
Main mem (GB)	3	4	4	3
Instruction Set	64-bit	64-bit	64-bit	64-bit
Turbo Boost	No	No	No	No
Hyper-Threading	No	No	No	No
Linux kernel	2.6.38.15	2.6.28.10	2.6.38.2	2.6.28.0
Ubuntu release	11.04	9.04	10.04	9.04
Perfctr patch	No	Yes	No	Yes

Table A.1: Experiment hosts

Appendix B

Benchmark descriptions

Benchmark	Description
Array creation	Measure the performance of creating arrays of integers, long, float and Object
Exception handling	Measure the performance of exception throwing for default and user defined exceptions
Generic	Measure the performance of generic code to increase a counter
Loop	Measure the performance of for/while loop constructs
Method calling	Measure the performance of calling normal/synchronised instance/static methods
Object creation	Measure the performance of creating objects of classes with various fields
Thread	Measure the performance of creating spawning and joining threads
EP	NAS EP benchmark to generate 16M normally distributed random numbers
FFT	256-point Fast Fourier Transform (and inverse)
Fibonacci	Calculate the 40th Fibannaci Number
Hanoi	Solve the 25 disk Tower of Hanoi problem
Sieve	Use sieve of Erasthosthenes to find prime numbers up to 10.000
IS	NAS Integer Sort Benchmark for 1M integers

Table B.1: DHPC sequential JGF benchmark descriptions

Benchmark	Description
SimpleBarrier	Measure throughput of synchronising threads on a barrier using monitors
TournamentBarrier	Measure throughput of synchronising threads on a barrier using Thread.yield
MethodSync	Synchronise threads on a synchronised method
ObjectSync	Synchronise threads on an object
ForkJoin	Measure throughput of creating and joining threads
Crypt	IDEA encryption
LUFact	LU Factorisation
Moldyn	Molecular Dynamics simulation
MonteCarlo	Monte Carlo simulation
RayTracer	3D Ray Tracer
Series	Fourier coefficient analysis
SOR	Successive over-relaxation method
SparseMatMult	Sparse matrix multiplication

Table B.2: JGF multi-threaded benchmark descriptions

Benchmark	Description	Threaded	I/O
compiler.compiler	Compile the OpenJDK compiler	1/core	Limited due to input data in memory
compiler.sunflow	Compile the sunflow benchmark	1/core	Limited due to input data in memory
compress	Compress and decompress data using a modified Lempel-Ziv method	1/core	Data buffered
crypto.aes	Encrypt and decrypt using AES	1/core	Reading input files
crypto.rsa	Encrypt and decrypt using DES	1/core	Reading input files
crypto.signverify	Sign and verify using md5 and sha hashing	1/core	Reading input files
derby	Derby DB performing BigDecimal computations	4 threads per db	I/O in communication with DB
mpegaudio	Mpeg audio decoder	1/core	Read 6 MP3 files
scimark.fft.large	Fast Fourier Transformation for 2MB data set	1/core	No
scimark.fft.small	Fast Fourier Transformation for 512KB data set	1/core	No
scimark.lu.large	LU factorisation on a dense 2048 x 2048 matrix	1/core	No
scimark.lu.small	LU factorisation on a dense 100 x 100 matrix	1/core	No
scimark.monte_carlo	Approximate value of pi using Monte Carlo simulation	1/core	No
scimark.sor.large	Jacobi Successive Over Relaxation for 2048x2048 grid	1/core	No
scimark.sor.small	Jacobi Successive Over Relaxation for 250x250 grid	1/core	No
scimark.sparse.large	Sparse matrix multiplication of a 200k x 200k matrix with 4m non-zero elements	1/core	No
scimark.sparse.small	Sparse matrix multiplication of a 25k x 25k matrix with 62.5k non-zero elements	1/core	No
serial	Serialise and de-serialise objects	1/core	No
sunflow	Use ray tracing to simulate graphics and visualisation	4/core	No
xml.transform	Compile xsl style sheets into java classes	1/core	Reading 10 input files
xml.validation	Validate XML instance documents against XML schema	1/core	Reading 6 input files

Table B.3: SPECjvm2008 benchmark descriptions

Appendix C

Framework usage

In this section we briefly present how VEX and JINE are used on a hypothetical class `Main.class`. If the generic form that `Main` is executed on the JVM is:

```
java -cp <classpath> Main <args>
```

where <args> are the space-separated arguments of the class, then VEX simulation can be triggered by:

```
java -cp <classpath> -Xbootclasspath/p:vtf.jar
   -javaagent:jine.jar=<JINE options>
   -agentpath:libjine.so=<VEX options> Main <args>
```

where `vtf.jar` is the jar file with the statically instrumented Java classes for the specific JVM, `jine.jar` is the Java-agent (with options appearing in Table C.2) and `libjine.so` the low-level JVMTI-agent that links to VEX (with the options of Table C.1). The classpath of `libjine.so` needs to be included in the `$LD_LIBRARY_PATH` environmental variable. Although the directories that contain the shared libraries used by `libjine.so` (amongst which `libvex.so`) are set during the compilation of the library, it might be worth checking that all dependencies are correctly resolved by using the `ldd` and `ld` linker tools. Any missing paths should be included in the `$LD_LIBRARY_PATH`.

C.1 Building and Options

C.1.1 VEX

The provided `Makefile.inc` file has to be modified for the local installation to link to the PAPI, libunwind and GSL libraries. The compilation has been tested with gcc versions 4.3.3, 4.4.3, 4.6.1 and 4.6.3, PAPI 4.1.0 (configuring with the "–with-perfctr", the "–with-perf-events" or none), libunwind 0.99 and GSL 1.15. The compilation generates three shared libraries: the `libcinqs.so` for the Queueing Networks simulator, the `libvex.so` for VEX and the `libjine.so` for the low-level JINE.

C.1.2 JINE

The provided *ant* file is responsible for building both `vtf.jar` and `jine.jar`. To build the jar of statically instrumented system classes for a particular JVM `vtf.jar`, the `JAVA_HOME` environmental variable has to be set to the home directory of that JVM installation. The jar is created by typing:

```
# export JAVA_HOME=<path to JDK>
# ant create_static_classes
```

in the same directory as the `build.xml` file provided.

Following this, the JINE Java-agent `jine.jar` is built by:

```
# ant
```

Accepted parameters in form $< parameter > [=< value >], < parameter > [=< value >], ..., < parameter > [=< value >]$	
VEX scheduler parameters	
cpus=value	Number of virtual cores to simulate
delays_file=filename	File with delays incurred by the instrumentation code. Otherwise using .vex_delays file. Otherwise profiling delays at runtime
help	Print this parameter menu and exit
no_recursive_change	Do not re-adapt recursive calls
no_scheduling	Do not schedule the threads in virtual time. Just profile their times
profile_yield	Profile the duration of *sched_yield*
spex	Enforce SPEX multicore synchronisation scheme [default: off]
timeslot=value	Timeslot duration of the scheduler. If value < 10 000 then timeslot=value ms (1ms to 10sec) otherwise timeslot=value ns
version	Print version of VEX before starting simulation
VEX - Java specific parameters	
exclude_threads=filename	File with the names of threads to be excluded
natwait=value	Identification method for native waiting threads. Values: none — vexStats — stackTrace — combined [default: vexStats]
no_gc_time	Do not take into account the duration of garbage collections in method times
no_jvmti	Explicit monitor trapping: Do not rely on JVMTI to track monitor changes
poll_nw	Poll native waiting threads to determine whether they should be put back into the framework
VEX output parameters	
output_dir=dir	Directory where output files will be written. [default: ./]
output_format=value	Format of output. Values: vtf—hprof [default: vtf]
print_recursion	Runtime output of recursive method names that will be re-adapted
profiling_HPC	Profile hardware performance counters (including VEX overhead) into $< output\_dir > /vtf\_cpi\_and\_cache\_stats.csv$
prune_percent	Methods with estimated real time (inclusive) less than prune_percent will not be reported [default: disabled]
scheduler_stats	Print statistics for thread states found by the scheduler into $< output\_dir > /vtf\_scheduler\_stats.csv$
stack_trace_mode	Store results in stack trace mode (not only according to method names) - use only in combination with with JINE's stackTraceMode
time_stats=value	Print time increase statistics into $< output\_dir > /vtf\_time\_stats.csv$. Values: global — thread — samples [default: global]
visualize	Log the visualization information into $< output\_dir > /vtf\_events.csv$
I/O handling parameters	
io=value	I/O method used. Values: serial — strict — normal — lax [default: normal]
iobuffer=size	Size of I/O prediction buffer [default: 16]
iopred=value	I/O prediction method. Values: replay — avg — min — max — median — sampling — gsl — cmean — linregr — aryulwalk — arburg — markov [default: avg-16]
iolevel=value	Select the level of I/O prediction. Values: none — global — invpoint — ioop (invocation point) [default: invpoint]
ioexclude=value	Random exclusion of value % I/O operations measuring their duration in CPU time [default: off]
iodisregard	Opt-out all I/O operations. Only their CPU time is taken into account. Same as ioexclude=100. [default: off]
iothreshold=value	I/O operations less than value ns will not be recognized as I/O but still measured in real time [default: off]
iostats=value	Generate I/O prediction statistics on the defined level. Values: off — global — invpoint — all [default: off]
iolog=value	Enable extra I/O logging options (iostats enables basic ones). Values: ext — gsl [default: off]
ioimport=file_prefix	Import I/O measurements from file_prefix into the current predictor [default: off]
ioexport=dir	Export I/O measurements from current predictor into directory dir [default: off]
no_io_reporting	Do not report I/O method times

Table C.1: VEX options

adaptedMethods=< *file* >	a file containing the methods that should be adapted together (time scaling factor/describing performance model)
adaptiveProfiling< *value* >	set invalidation policy. value can be "=samples" (per method), ":int absolute ms" (per method) or ":float relative time" (per method)
externalInstrumentation	put profiling instruments externally of each profiled method to avoid changes in JIT-compiler
help	print this menu and exit
limitJvmti	do not use JVMTI to trap MonitorWait(ed) and MonitorContendedEnter(ed) events explicitly instrument the calls
limitProfiling=< *value* >	limit the adding of profiling instruments. Value=none
lineLevelStackTrace	I/O invocation points differentiate according to the calling methods lines
methodFile=< *file* >	a file containing the methods that should be instrumented
noModifyTimeCalls	do not modify calls System.nanoTime() and System.currentTimeMillis() to return simulation timestamps
noWait	do not instrument Object.wait calls
outputInstrClasses=< *dir* >	output the instrumented classes bytecode under the specified directory
package=< *file* >	a file containing the packages that should be instrumented
printoutMethodsFile=< *file* >	a file containing classes and methods that should print out a message when executed (for dbg reasons)
profileHost=option	profile the instrumentation delays on this host and store them to .vex_delays or file of delays_file VEX parameter option can be min — avg — median
profilingRun	do not instrument the synchronised keywords
removeMonitors	(only effective with limitJmvti): replace monitors by low-level locks
stats=< *value* >	output instrumentation statistics to */data/jine_instrumentation_statistics.csv*. Value = global — class — method
trapIPs	trap Interaction Points

Table C.2: JINE options

Bibliography

[1] JUnitPerf. http://www.clarkware.com/software/JUnitPerf.html.

[2] The DiskSim Simulation Environment (V4.0). http://www.pdl.cmu.edu/DiskSim/, December 2011.

[3] E. Anderson. Simple table-based modeling of storage devices. Technical report, Tech Rep. HPL-SSP-2001-04. HP Laboratories, 2001.

[4] Apache Software Foundation. Apache Derby. http://db.apache.org/derby/.

[5] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43:52–61, January 2009.

[6] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35:59–67, February 2002.

[7] R. Bagrodia, E. Deeljman, S. Docy, and T. Phan. Performance prediction of large parallel applications using parallel simulations. In *Proceedings of the 7th ACM symposium on Principles and practice of parallel programming*, PPoPP '99, pages 151–162, New York, NY, USA, 1999. ACM.

[8] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, 2004.

[9] S. Balsamo and M. Marzolla. Simulation modeling of UML software architectures. In *Proc. of ESM03, the 17th European Simulation Multiconference*, pages 562–567. SCSEuropean Publishing House, 2003.

[10] N. Baltas and T. Field. Software performance prediction with a time scaling scheduling profiler. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2011. MASCOTS 2011. IEEE 19th International Symposium on*, pages 107 –116, July 2011.

[11] N. Baltas and T. Field. Continuous performance testing in virtual time. In *Quantitative Evaluation of Systems (QEST), 2012 Ninth International Conference on*, pages 13 –22, September 2012.

[12] J. R. Bammi, W. Kruijtzer, L. Lavagno, E. A. Harcourt, and M. T. Lazarescu. Software performance estimation strategies in a system-level design tool. In *CODES'00*, pages 82–86, 2000.

[13] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[14] C. Bergstrom, S. Varadarajan, and G. Back. The distributed open network emulator: Using relativistic time for distributed scalable simulation. *Parallel and Distributed Simulation, Workshop on*, 0:19–28, 2006.

[15] S. Bernardi and D. C. Petriu. Comparing two UML profiles for non-functional requirement annotations. In *the SPT and QoS Profiles, UML'2004*.

[16] M. Bertoli, G. Casale, and G. Serazzi. JMT: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009.

[17] G. Bolch, S. Greiner, H. de Meer, and K. S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications*. WileyBlackwell, 2nd edition, May 2006.

[18] R. Bounds. Virtual time execution. *Imperial College MSc Thesis*, October 2008.

[19] P. Bourke. AutoRegression analysis. http://paulbourke.net/miscellaneous/ar/.

[20] D. P. Bovet and M. C. Ph. *Understanding the Linux Kernel, Third Edition.* O'Reilly Media, 3 edition, November 2005.

[21] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical report, 1991.

[22] E. Bruneton. ASM 3.0, a Java bytecode engineering library. http://download.forge.objectweb.org/asm/asm-guide.pdf.

[23] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[24] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Addison-Wesley Publishing Company, USA, 2nd edition, 2010.

[25] H. W. Cain, K. M. Lepak, B. A. Schwartz, and Mikko. Precise and accurate processor simulation. *In Proceedings of the Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*, 2002.

[26] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 340 – 351, February 2005.

[27] K. M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[28] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, April 1981.

[29] J. Chen, M. Annavaram, and M. Dubois. SlackSim: a platform for parallel simulations of CMPs on CMPs. *SIGARCH Comput. Archit. News*, 37:20–29, July 2009.

[30] Compuware. Applied performance management survey. October 2006.

[31] V. Cortellessa, A. D. Marco, and P. Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.

[32] R. C. Covington, S. Madala, V. Mehta, J. R. Jump, and J. B. Sinclair. The Rice parallel processing testbed. In *Proceedings of the 1988 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 4–11, New York, NY, USA, 1988. ACM.

[33] M. Dahm. Byte code engineering. http://jakarta.apache.org/bcel/, September 1999.

[34] David Mosberger. The libunwind project page. http://www.nongnu.org/libunwind/people.html.

[35] H. Davis, S. R. Goldschmidt, and J. L. Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *ICPP (2)'91*, pages 99–107, 1991.

[36] G. Denaro, A. Polini, and W. Emmerich. Early performance testing of distributed software applications. *SIGSOFT Softw. Eng. Notes*, 29(1):94–103, 2004.

[37] M. Dmitriev. Selective profiling of Java applications using dynamic bytecode instrumentation. In *Performance Analysis of Systems and Software, 2004. ISPASS 2004. IEEE International Symposium on*, pages 141 – 150, 2004.

[38] 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.*, 33(5):261–276, 1999.

[39] N. Dunn. Performance evaluation of semi-complete software. *Imperial College MSc Thesis*, July 2007.

[40] S. Dwarkadas, J. R. Jump, and J. B. Sinclair. Execution-driven simulation of multiprocessors: address and timing analysis. *ACM Trans. Model. Comput. Simul.*, 4(4):314–338, 1994.

[41] M. B. Dwyer, A. Kinneer, and S. Elbaum. Adaptive online program analysis. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 220–229, Washington, DC, USA, 2007. IEEE Computer Society.

[42] ej-technologies. JProfiler product page. http://www.ej-technologies.com/products/jprofiler/overview.html.

[43] M. A. Erazo and J. Liu. A model-driven emulation approach to large-scale TCP performance evaluation. *Int. J. Commun. Netw. Distrib. Syst.*, 5:130–150, July 2010.

[44] A. Eustace and A. Srivastava. Atom: a flexible interface for building high performance program analysis tools. *Proceedings of the USENIX 1995 Technical Conference Proceedings*, pages 25–25, 1995.

[45] T. Field. JINQS: An extensible library for simulating multiclass queueing networks, v1.0 User Guide. http://www.doc.ic.ac.uk/ ajf/Research/manual.pdf.

[46] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. Mock roles, not objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 236–246, New York, NY, USA, 2004. ACM.

[47] R. Fujimoto. *Parallel and distributed simulation systems*. Wiley series on parallel and distributed computing. Wiley, 2000.

[48] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. *GNU Scientific Library: Reference Manual*. Network Theory Ltd., February 2003.

[49] D. Garcia and J. Garcia. TPC-W e-commerce benchmark evaluation. *Computer*, 36(2):42 – 48, February 2003.

[50] J. Garcia, L. Prada, J. Fernandez, A. Nunez, and J. Carretero. Using black-box modeling techniques for modern disk drives service time simulation. In *Simulation Symposium, 2008. ANSS 2008. 41st Annual*, pages 139 –145, April 2008.

[51] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 57–76. ACM, 2007.

[52] S. R. Goldschmidt and J. L. Hennessy. The accuracy of trace-driven simulations of multiprocessors. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '93, pages 146–157, New York, NY, USA, 1993. ACM.

[53] A. Grau, K. Herrmann, and K. Rothermel. Efficient and scalable network emulation using adaptive virtual time. In *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*, pages 1–6. IEEE, 2009.

[54] A. Grau, S. Maier, K. Herrmann, and K. Rothermel. Time jails: A hybrid approach to scalable network emulation. In *PADS '08: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 7–14, Washington, DC, USA, 2008. IEEE Computer Society.

[55] N. J. Gunther. *Analyzing computer system performance with Perl::PDQ*. Springer, 2005.

[56] D. Gupta, K. V. Vishwanath, and A. Vahdat. Diecast: testing distributed systems with an accurate scale model. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 407–422, Berkeley, CA, USA, 2008. USENIX Association.

[57] 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 20th ACM Symposium on Operating Systems Principles*, pages 1–2, New York, NY, USA, 2005. ACM.

[58] S. Hammond, G. Mudalige, J. Smith, S. Jarvis, J. Herdman, and A. Vadgama. WARPP: a toolkit for simulating high-performance parallel scientific codes. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–10. ICST (In-

stitute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2009.

[59] J. Hillston. *A Compositional Approach to Performance Modelling*. Cambridge University Press, 1996.

[60] J. Hillston. Process algebras for quantitative analysis. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 239–248, Washington, DC, USA, 2005. IEEE Computer Society.

[61] C. E. Hrischuk, C. M. Woodside, J. A. Rolia, and R. Iversen. Trace-based load characterization for generating performance software models. *IEEE Transactions on Software Engineering*, 25(1):122–135, January/February 1999.

[62] W. Hsu and A. J. Smith. The performance impact of I/O optimizations and disk improvements. *IBM J. Res. Dev.*, 48:255–289, March 2004.

[63] W.-C. Hsu, S.-H. Hung, and C.-H. Tu. A virtual timing device for program performance analysis. In *Proceedings of the 2010 10th IEEE Intnl Conference on Computer and Information Technology*, CIT '10, pages 2255–2260, Washington, DC, USA, 2010. IEEE Computer Society.

[64] IBM. IBM Rational PurifyPlus, Purify, PureCoverage, and Quantify: Getting Started. http://www-01.ibm.com/software/awdtools/purifyplus/, May 2002.

[65] Intel. Intel VTune amplifier XE product page. http://software.intel.com/en-us/intel-vtune-amplifier-xe.

[66] T. A. Israr, D. H. Lau, G. Franks, and M. Woodside. Automatic generation of layered queuing software performance models from commonly available traces. In *Proceedings of the 5th international workshop on Software and performance*, WOSP '05, pages 147–158, New York, NY, USA, 2005. ACM.

[67] R. Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling*. Wiley, New York, 1991.

[68] Java Grande Forum. Multi-threaded benchmarks. http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/index_1.html.

[69] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 77–93, New York, NY, USA, 1987. ACM.

[70] D. R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[71] M. Johnson. Monitoring and diagnosing application response time with ARM. In *Systems Management, 1998. Proceedings of the IEEE Third International Workshop on*, pages 4–13, Apr 1998.

[72] N. Joukov, T. Wong, and E. Zadok. Accurate and efficient replaying of file system traces. In *In Proceedings of the USENIX Conference on File and Storage Technologies (FAST05)*, pages 337–350. USENIX Association, 2005.

[73] JUnit. JUnit.org Resources for TDD. http://www.junit.org/.

[74] T. Kelly, I. Cohen, M. Goldszmidt, and K. Keeton. Inducing models of black-box storage arrays. Technical report, 2004.

[75] T.-W. H. Kuo-Yi Chen, J.Morris Chang. Multi-threading in Java: Performance and scalability on multi-core systems. *IEEE Transactions on Computers*, 2009.

[76] A. M. Law and D. W. Kelton. *Simulation Modelling and Analysis*. McGraw-Hill Education - Europe, April 2000.

[77] M. T. Lazarescu, J. R. Bammi, E. Harcourt, L. Lavagno, and M. Lajolo. Compilation-based software performance estimation for system level design. In *Proceedings of the IEEE International High-Level Validation and Test Workshop (HLDVT'00)*, HLDVT '00, pages 167–, Washington, DC, USA, 2000. IEEE Computer Society.

[78] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[79] S. Li and H. Huang. Black-box performance modeling for solid-state drives. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS), 2010 IEEE International Symposium on*, pages 391 –393, aug. 2010.

[80] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[81] H. H. Liu. *Software Performance and Scalability: A Quantitative Approach*. Wiley Publishing, 2009.

[82] B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Commun. ACM*, 32(1):111–123, Jan. 1989.

[83] E. Machkasova, K. Arhelger, and F. Trinciante. The observer effect of profiling on dynamic Java optimizations. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 757–758, New York, NY, USA, 2009. ACM.

[84] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35:50–58, February 2002.

[85] F. Mallet and R. de Simone. MARTE: a profile for RT/E systems modeling, analysis– and simulation? In *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, pages 1–8, ICST, Brussels, Belgium, Belgium, 2008. ICST.

[86] A. Malony, D. Reed, and H. Wijshoff. Performance measurement intrusion and perturbation analysis. *Parallel and Distributed Systems, IEEE Transactions on*, 3(4):433 –450, July 1992.

[87] A. D. Malony and S. S. Shende. Overhead compensation in performance profiling. In *Proceedings of the 10th International Euro-Par Conference on Parallel Processing (Euro-Par 04*, pages 119–132. Springer-Verlag, 2004.

[88] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, P. Computing, J. Wiley, V. Almeida, J. Almeida, and C. M. P. Analysis. Modelling with Generalized Stochastic Petri Nets. In *Series in Parallel Computing*. John Wiley & Sons, 1995.

[89] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33:92–99, November 2005.

[90] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. *SIGMETRICS Perform. Eval. Rev.*, 30:108–116, June 2002.

[91] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea. *Performance testing guidance for web applications: patterns & practices*. Microsoft Press, Redmond, WA, USA, 2007.

[92] M. Meswani, M. Laurenzano, L. Carrington, and A. Snavely. Modeling and predicting disk I/O time of HPC applications. In *High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2010 DoD*, pages 478 –486, June 2010.

[93] T. Meyerowitz, D. Langen, M. Sauermann, and A. Sangiovanni-Vincentelli. Source-level timing annotation and simulation for a heterogeneous multiprocessor. In *Design Automation Test Europe*. IEEE, March 2008.

[94] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[95] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1 –12, jan. 2010.

[96] A. Mizan and G. Franks. An automatic trace based performance evaluation model building for parallel distributed systems. In *ICPE*, pages 61–72, 2011.

[97] I. Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 1st edition, 2009.

[98] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, M. Hill, D. Wood, S. Huss-Lederman, and J. Larus. Wisconsin Wind Tunnel II: a fast, portable parallel architecture simulator. *Concurrency, IEEE*, 8(4):12 –20, Oct-Dec 2000.

[99] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44:265–276, March 2009.

[100] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the accuracy of Java profilers. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 187–197, New York, NY, USA, 2010. ACM.

[101] H. Nguyen and A. Apon. Hierarchical performance measurement and modeling of the linux file system. In *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering*, ICPE '11, pages 73–84, New York, NY, USA, 2011. ACM.

[102] J. Oly and D. A. Reed. Markov model prediction of I/O requests for scientific applications. In *Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 147–155, New York, NY, USA, 2002. ACM.

[103] OMG. UML profile for schedulability, performance and time specification, Final Adopted Specification ptc/02-03-02, March 2002.

[104] OMG. The UML profile for modeling and analysis of real-time and embedded systems, Beta 1. OMG adopted specification, ptc/07-08-04, 2005.

[105] OMG. UML profile for modeling QoS and FT characteristics and mechanisms v1.1. Technical report, Object Management Group, April 2008.

[106] A. Over, B. Clarke, and P. Strazdins. A comparison of two approaches to parallel simulation of multiprocessors. In *Performance Analysis of Systems and Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 12 –22, April 2007.

[107] H. Pan, K. Asanović, R. Cohn, and C.-K. Luk. Controlling program execution through binary instrumentation. *SIGARCH Comput. Archit. News*, 33:45–50, December 2005.

[108] PAPI. Performance application programming interface website. http://icl.cs.utk.edu/papi/index.html.

[109] A. Patel, F. Afram, S. Chen, and K. Ghose. MARSSx86: A full system simulator for x86 CPUs. In *Design Automation Conference 2011 (DAC'11)*, 2011.

[110] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Softw. Pract. Exper.*, 37(7):747–777, 2007.

[111] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. Pin: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*, New York, NY, USA, 2004. ACM.

[112] P. L. Reiher and D. Jefferson. Virtual time based dynamic load management in the Time Warp Operating System. *Transactions of the Society for Computer Simulation*, 7:103–111, 1990.

[113] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '93, pages 48–60, New York, NY, USA, 1993. ACM.

[114] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Comput. Commun. Rev.*, 27:31–41, January 1997.

[115] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. D. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The structural simulation toolkit. *SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[116] J. Rolia and K. Sevcik. The method of layers. *IEEE Transactions on Software Engineering*, 21:689–700, 1995.

[117] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: the SimOS approach. *Parallel Distributed Technology: Systems Applications, IEEE*, 3(4):34 –43, Winter 1995.

[118] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Comput. Archit. Lett.*, 10(1):16–19, Jan. 2011.

[119] B. Scott. User experience, not metrics. Published in website: http://www.ibm.com/developerworks/rational/library/4228.html, May 2004.

[120] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 42:1–42:12, Piscataway, NJ, USA, 2008. IEEE Press.

[121] K. Shen, C. Stewart, C. Li, and X. Li. Reference-driven performance anomaly identification. In *Proceedings of the 2009 ACM SIGMETRICS conference on Measurement and modeling of computer system*, SIGMETRICS '09, pages 85–96, New York, NY, USA, 2009. ACM.

[122] K. Shen, M. Zhong, and C. Li. I/O system performance debugging using model-driven anomaly characterization. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies - Volume 4*, pages 23–23, Berkeley, CA, USA, 2005. USENIX Association.

[123] J. Shirazi. *Java Performance Tuning*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[124] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 performance characterization. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35, Berlin, Heidelberg, 2009. Springer-Verlag.

[125] C. Smith and L. Williams. *Performance solutions: a practical guide to creating responsive, scalable software*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[126] M. Smith. Towards stochastic model extraction: Performance evaluation, fresh from the source. In *Proceedings of Process Algebra and Stochastically Timed Activities (PASTA) 2006*, 2006.

[127] T. Software. TIOBE Programming Community Index for September 2012. http://http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[128] Standard Performance Evaluation Corporation (SPEC). Specjvm2008 benchmarks.

[129] Sun Microsystems. JVM tool interface specification version 1.1. http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html.

[130] Sun Microsystems. Java native interface specification. http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html, May 1997.

[131] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

[132] P. Trapp and C. Facchi. How to Handle CPU Bound Systems: A Specialization of Dynamic Performance Stubs to CPU Stubs. In *CMG 08: International Conference Proceedings*, pages 343 – 353. Computer Measurement Group, 2008.

[133] D. Tsafrir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *ACM Workshop on Experimental Computer Science (ExpCS)*, page 4, San-Diego, California, June 2007.

[134] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29:128–170, 2004.

[135] University of Adelaide, Distributed and High-Performance Computing Group. Java Grande Benchmarks. http://www.dhpc.adelaide.edu.au/projects/javagrande/benchmarks/.

[136] S. Varadarajan. The weaves reconfigurable programming framework. http://www-01.ibm.com/software/awdtools/purifyplus/, 2002.

[137] K. Wang, Y. Zhang, H. Wang, and X. Shen. Parallelization of IBM mambo system simulator in functional modes. *SIGOPS Oper. Syst. Rev.*, 42:71–76, January 2008.

[138] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. Ganger. Storage device performance prediction with cart models. In *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on*, pages 588 – 595, October 2004.

[139] Z. Wang and A. Herkersdorf. Software performance simulation strategies for high-level embedded system design. *Performance Evaluation*, 67(8):717–739, 2010.

[140] Z. Wang, R. Liu, Y. Chen, X. Wu, H. Chen, W. Zhang, and B. Zang. COREMU: a scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 213–222, New York, NY, USA, 2011. ACM.

[141] V. M. Weaver and J. Dongarra. Can hardware performance counters produce expected, deterministic results? In *FHPM-2010: The 3rd Workshop on Functionality of Hardware Performance Monitoring*, December 2010.

[142] E. Weingärtner, F. Schmidt, T. Heer, and K. Wehrle. Synchronized network emulation: matching prototypes with complex simulations. *SIGMETRICS Perform. Eval. Rev.*, 36(2):58–63, 2008.

[143] E. Weingärtner, F. Schmidt, H. V. Lehn, T. Heer, and K. Wehrle. SliceTime: a platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 19–19, Berkeley, CA, USA, 2011. USENIX Association.

[144] C. C. Williams and J. K. Hollingsworth. Interactive binary instrumentation. In *Int'l. Works. on Remote Anal. and Measurement of Softw. Sys.*, May 2004.

[145] E. Witchel and M. Rosenblum. Embra: fast and flexible machine simulation. In *Proceedings of the 1996 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, SIGMETRICS '96, pages 68–79, New York, NY, USA, 1996. ACM.

[146] M. Woodside, G. Franks, and D. Petriu. The future of software performance engineering. *Future of Software Engineering (FOSE '07)*, pages 171–187, May 2007.

[147] M. Woodside and D. Petriu. Capabilities of the UML profile for schedulability performance and time. In *SPT Workshop SIVOES-SPT RTAS'2004*, 2004.

[148] T. Yoshida, H. Yamada, and K. Kono. FoxyLargo: Slowing down CPU speed with a virtual machine monitor for embedded Time-Sensitive software testing. In *International Workshop on Virtualization Technology (IWVT'08)*, Beijing, China, 2008.

[149] M. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems and Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23 –34, April 2007.

[150] D. Zaparanuks, M. Jovic, and M. Hauswirth. Accuracy of performance counter measurements. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 23 –32, April 2009.

[151] J. Zhang and R. J. O. Figueiredo. Adaptive predictor integration for system performance prediction. In *IPDPS*, pages 1–10. IEEE, 2007.

[152] G. Zheng, G. Kakulapati, and L. Kale. BigSim: a parallel simulator for performance prediction of extremely large parallel machines. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 78, April 2004.