Continuous Performance Testing in Virtual Time

Nikos Baltas
Department of Computing
Imperial College London 180 Queen’s Gate
London SW7 2AZ
Email: nb605@doc.ic.ac.uk

Tony Field
Department of Computing
Imperial College London 180 Queen’s Gate
London SW7 2AZ
Email: ajf@doc.ic.ac.uk

Abstract—In this paper we show how program code and performance models can be made to cooperate seamlessly to support continuous software performance testing throughout the development lifecycle. We achieve this by extending our existing VEX tool for executing programs in virtual time so that events that occur during normal execution and those that occur during the simulation of a performance model can be scheduled on a single global virtual time line. The execution time of an incomplete component of an application is thus estimated by a performance model, whilst that of existing code is measured by instrumentation that is added dynamically at program load time. A key challenge is to be able to map some or all of the resources in a performance model to the real resources of the host platform on which the application is running. We outline a continuous performance engineering methodology that exploits our unified framework and illustrate the principles involved by way of a simple Java application development case study.

I. INTRODUCTION

In this paper, we present a new approach to software performance engineering that supports continuous software performance testing throughout the development lifecycle. 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.

Smith and Williams were the first to propose a comprehensive lifecycle-wide performance prediction approach in Software Performance Engineering (SPE) [1]. In their work, performance modelling techniques are used during the early stages and empirical methods are used in the late stages of the lifecycle. During intermediate stages, measurements on real code are used to verify or calibrate the performance models, which are solved using a variety of methods to provide performance predictions. The methodology we propose here is designed to streamline this SPE process, allowing models and code to co-exist within the same framework. We also circumvent the problems associated with modelling existing code. Here, we don’t model the code; we execute it!

We implement our methodology via an extension to the VEX framework [2], which constitutes a low-level scheduler that virtualizes the passing of time for multi-threaded Java applications. VEX is essential to our approach, because it enables virtual execution times from the simulation of models to be mapped together with real execution-time measurements onto a single global virtual timeline. In principle, any formalism can be used to define a performance model, but for the purposes of this paper we will focus on (open) queueing networks that are specified using the Java Modelling Tools (JMT) [3] and simulated in virtual time using a separate discrete-event simulation tool. The virtual time spent in a queuing 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 queuing 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 our 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 [2]. The simulation models resource consumption and queuing, 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.

Our 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 [4].

The contributions of the paper are as follows:

- We present a methodology for integrating performance models and real code measurements in virtual time to enable end-to-end performance testing throughout the
software development lifecycle (Section III)

- We extend VEX’s modelling capabilities so that it can model contention for both local and remote resources (Section III-D). In particular, virtual resources in a performance model can be associated with real resources on the host platform.

- We detail the extensions to the scheduling profiler framework and the JMT suite that were required to support the integration of queueing networks and executing code (Section IV).

- We demonstrate the performance engineering methodology that our framework facilitates by documenting a sample scenario – the development of client-server application based on a MySQL database (Section V)

As part of the future work (Section VII) we discuss how the framework can be integrated with existing simulators.

II. BACKGROUND

The VEX framework [2] is an execution-driven simulator, that investigates the performance impact of post-implementation method-level optimization alternatives. By controlling the progress of application threads, VEX emulates the behaviour that the application would exhibit, if specific user-definable method time-scaling factors were applied. We refer to this kind of simulation as a virtual-time execution.

We illustrate the implications of virtual-time execution through the example of Figure 1. This shows the progress of two threads, $T_1$ and $T_2$, when they are executed on a single-core machine. 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 schedule results into both threads issuing I/O requests in parallel. However, if we accelerate by a factor of two the execution of $T_2$ (or, more specifically, the method being executed by $T_2$) prior to requesting the lock, then a new interleaving of the threads is produced: $T_2$ acquires the lock first, releases it within the same timeslice, and then allows $T_1$ to make progress immediately. $T_2$ is now the first thread to issue an I/O request, which finishes before $T_1$ issues its own. VEX is designed to simulate correctly this new virtual behaviour of the program.

A VEX scheduler enforces a schedule that is in accordance with the defined time-scaling factors, by deciding the duration of the timeslice of each application thread. The higher the acceleration factor of the method that a thread is executing, the longer the scheduler allows this thread to continue. Once the timeslice of this thread expires, VEX adds the time-scaled CPU-time duration of its execution to the global virtual time of the simulation. For example, if a thread is accelerated by a factor of 2, then it is allowed to execute for twice the duration of the regular VEX timeslice and has its profiled CPU-time be divided by two to reflect the speedup. VEX guarantees a fair (round-robin) and consistent thread interleaving, by always resuming the runnable thread that was suspended at the lowest global virtual time. To distinguish runnable from waiting or blocked threads, the framework monitors thread-state changes triggered by semaphore contentions and blocking I/O operations.

The key idea of VEX is to model method acceleration by creating the illusion that the thread executing the method is progressing faster (or slower) than the other threads for the duration of the method execution. It does this by scaling the measured execution time according to a time-scaling factor associated with that method. Although this approach seems to oversimplify the performance effect of the acceleration, we underline here that the effects of the virtual-time schedule on the application and system resources are still reflected in the measured durations. In this paper, we extend VEX’s modelling capabilities, by entirely decoupling the duration of the thread execution on the simulation host from its virtual-time performance. Its expected performance is now defined by simulating a performance model – in this case an open queueing network 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 software life cycle, using stubs and mock objects [5].

The work presented here uses the thread-state monitoring and scheduling, instrumentation and method profiling capabilities of VEX. However, all model handling approaches regarding mainly the definition of models (in Section IV-A), the monitoring of state-changes and inclusion of model simulation times (in IV-C) and the synchronisation of real code with model simulations (see III-C) constitute new extensions to the original framework.

III. 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.

A. 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 time is advanced instantaneously from one scheduled event to the next.

We merge the two notions of time by mapping both of them onto a single virtual time line. This means that if an event is scheduled to happen $\Delta t$ in the future according to a model, then existing threads are assigned $\Delta t$ 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 $\Delta t$. By controlling the execution of threads based on thread-local (virtual) timestamps, a virtual time schedule that is consistent with both code execution and performance model simulation is maintained.

Crucially, a VEX-like simulation framework is necessary for transparently integrating models and code, by properly selecting and logging the execution and simulation times on its single virtual timeline. This 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. Apart from this, the role of the VEX remains to determine the schedule of the program threads by maintaining timestamps for each of them (executing real code or simulating models), within a timeslice distance (see [2] for details).

B. Merging workloads

The key idea here is that jobs within a model are associated with threads on an one-to-one basis. 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 2. 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 2 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 run-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. What matters to the VEX scheduler is virtual time. In Figure 2, 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 2), 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.

C. 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 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 3. 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 T1 executes the code of method M1, the VEX scheduler resumes another thread T2 which enters a synchronisation block on some object obj (2), before suspending when its timeslice expires (3). At this point, the model-predicted virtual timestamp of thread T1 is the smallest in the simulator so the scheduler waits for the completion of M before resuming T1. When T1 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.

In our prototype implementation 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 3 for T1, 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 T2 is allowed to resume (6). If no other runnable thread exists, then the deadlock is an artifact of the application itself. Allowing other threads to make progress will eventually lead to a real code event (in our 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 synchronization 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.

D. 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 characterized 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 lead the threads that “execute” them to block for an amount of time specified by the model. CPU resources remain available to any runnable (or local-resource simulating) threads.

IV. INTEGRATING OPEN QUEUEING NETWORKS WITH VEX

In this section we detail the tools and techniques used to implement our methodology for Java applications. The
performance models that we use are open queueing networks, although there is nothing in principle that precludes the use of other types of 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 III-B, 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 III-A). Routing decisions are determined by the model’s parameters. We handle queueing by setting the state of the thread Tj that corresponds to the queueing job as blocked. Since Tj 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 of our prototype implementation 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 [6], that builds on the popular JUnit [7] 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 for our case study.

A. Defining method-level performance models

The performance models that are supported by our prototype are queueing networks, generated by the open-source Java Modelling Tools (JMT) suite [3]. 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 III-D) 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) [2], 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:

```java
@virtual.time.ModelPerformance (jmtModelFilename = "models/qn.jsimg",
replaceMethodBody = false,
customerClass = "Class0",
sourceLabel = "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 [8]; 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, defers to VEX, which already supports this functionality.

B. Assumptions

Our prototype implementation does not support all of the features of either JMT or JINQS so our 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. For the purposes of this paper, none of these restrictions are significant and all of them can 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 [9], 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 VII.

C. 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 III-B. 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:

- If $J$ goes into service at a “remote” server (see Section III-D), 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 4), then the residual service time, $r$ say, is given by $s - t_{serv}$, where $s$ is the scheduler timeslice and $t_{serv}$ 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. 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.

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 5 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 6 (lower part). In contrast, jobs that are being serviced by remote resources can resume

![Figure 5](image-url)

![Figure 6](image-url)
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 6. 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 \( t_x - t_0 \).

V. CASE STUDY

We demonstrate our approach 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 our framework 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.

**What we do not do here:** 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 our 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 [1] 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 scaling observed measurements as described in [2].

Our approach 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 \( \lambda \). 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; our 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 [10]. 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 \( \rho = S_c/N \). \( \rho \) 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 I. All measurements have been taken on the 64-bit Hotspot(TM) (1.6.23.0-b07) JVM on an Intel(R) Core(TM) 2 Duo CPU E6750, 2.66GHz with 4GB of RAM, running on an 64-bit Ubuntu 9.04 kernel 2.6.24 patched with PerfCrt 2.6 for access to CPU-time counters. In this example scenario we only consider single CPU hosts and so the second core of the system is disabled. Dummy requests are issued prior to the actual measurement start to limit warm-up effects from the JVM. We use version 5.5 of the MySQL server.

A. 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 \texttt{think()} method of a \texttt{ThinkingBehaviour} interface. Initially, this method is described by a single infinite server node sampling from an exponential distribution with rate \( \lambda \). 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 \texttt{service()} method of a \texttt{ServiceBehaviour} interface. This method is originally simulated by the model of Figure 7. 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 our assumptions, the routing probabilities from the source to the two nodes are \( \rho \) for the “Local Cache” and \( 1 - \rho \) for the “MySQL server”.

The purpose of this section is to demonstrate the ability of our framework to allow models and code to

### TABLE I
PARAMETER VALUES AND DISTRIBUTIONS FOR THE CASE STUDY.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of requests, ( R )</td>
<td>3000</td>
</tr>
<tr>
<td>Request arrival rate, ( \lambda )</td>
<td>400</td>
</tr>
<tr>
<td>No. of request types, ( N )</td>
<td>52</td>
</tr>
<tr>
<td>Cache size, ( S_c )</td>
<td>39</td>
</tr>
<tr>
<td>MySQL service time distribution</td>
<td>Det(0.007)</td>
</tr>
<tr>
<td>Cache service time distribution</td>
<td>Exp(20000)</td>
</tr>
<tr>
<td>Cache hit rate, ( \rho )</td>
<td>0.75</td>
</tr>
</tbody>
</table>

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 II.

#### B. 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 \( \lambda \)), 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 Table II. 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 7.

#### C. 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 9. 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 10. This shows clearly how real code and performance models are integrated within our approach. 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 II.

#### D. Complete code

Completing the code base, we replace the body of the `makeRequestToDBserver()` method (code in Figure 11) to issue the request to the MySQL server running on a different host (with the same specifications) within our local area network (code in Figure 12). Executing the performance test in virtual time results in a slight increase in the expected times (fourth row of Table II). We regard this to be related to the overheads from real time I/O measurements, that also include some background system noise (see [2]). 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 II), 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% increase in the average think time of the real-time execution compared to the requested \( \lambda \) 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 13. 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 ‘artifacts’.

#### E. 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
TABLE II
TOTAL TIMES FOR PERFORMANCE TESTS IN THE DIFFERENT DEVELOPMENT STAGES OF OUR CASE STUDY (30 RUNS PER RESULT)

<table>
<thead>
<tr>
<th>Development stage</th>
<th>Time [ms]</th>
<th>Stdev [ms]</th>
<th>Error (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure model</td>
<td>7857.10</td>
<td>53.366</td>
<td>-16.53</td>
</tr>
<tr>
<td>Client-thinking implemented</td>
<td>7751.67</td>
<td>145.938</td>
<td>-17.85</td>
</tr>
<tr>
<td>Partially-complete code</td>
<td>7801.20</td>
<td>158.632</td>
<td>-17.13</td>
</tr>
<tr>
<td>VEX complete code</td>
<td>8576.93</td>
<td>123.045</td>
<td>-8.88</td>
</tr>
<tr>
<td>Real complete code</td>
<td>9413.07</td>
<td>140.633</td>
<td>0</td>
</tr>
</tbody>
</table>

Fig. 10. demo_select_only_db.jsimg: Queueing network for remote server only

total test time per cache size $S_c = 1...N$ for the “Real” and the “Partially-complete code” are illustrated in Figure 14. This presents a good prediction accuracy for our method 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 V-D above).

VI. RELATED WORK

The VEX framework follows the execution-driven simulation paradigm [4], where simulators are driven by real workloads. The functional behaviour is determined by running [4], [11], [12], [13] or emulating [14], [15] programs on the simulation host, whilst determining their timing-behaviour using either measurements [13], mathematical [12], [14] or simulation models [15]. Native-executed code drives performance models in our approach as well. The key difference is that the (local) virtual resources are shared by both models and code, thus creating a resource contention between the program’s execution and the non-functional simulation. This is accomplished by merging simulation time and real time on a new time dimension and coordinating the state changes resulting from real code and model simulation.

A similar approach is followed in the SliceTime [9] platform, which integrates network simulators and code prototypes. SliceTime 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. Though quite similar to our approach, the SliceTime implementation is based on executing software prototypes within virtual machines, by modifying the hypervisor layer to virtualize 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.

Smith and Williams [1] propose an approach to the integration of models and code, where measurements on implemented parts of the code are used to build new or calibrate existing models, that are then simulated to provide performance predictions. Though this approach is fast and generally applicable, it may exhibit low accuracy, by abstracting away implementation details of potential performance significance, or increase the analysis effort, by requiring the modelling of a potentially large or poorly understood code base.

A different approach is to try to “execute” models in real time as presented in [16]. The idea is to simulate models in real time, by forcing the thread that is executing a model-described method to sleep or busy-wait for a duration sampled by a modelling formalism. As existing methods consume CPU cycles as well, models and code are integrated in real time, leading them to contend for the same resource (CPU). However, contention on “remote” resources or queuing effects on “local” resources are not handled. The simulation has to occur in real time, increasing performance testing time. As the control and order of execution is still determined by the OS, the analysis capabilities of the approach are limited.

To the best of our knowledge the only published work which relates to performance testing of incomplete software, is proposed by Denaro et al in [17]. In this work performance testing on middleware-based software is applied early in its development lifecycle. The solution replaces unimplemented methods with stubs [5], whose expected performance is disregarded, under the assumption that the main performance results are based on the deployment of the out-of-the-shelf implemented components. Although this assumption applies for commercial off-the-shelf (COTS) based software, it does not cover arbitrary production code. 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.

VII. FUTURE WORK

We are planning to investigate the matching of queueing network nodes to system resources, like file systems or I/O sub-systems, by treating traffic to these resources as job
Job arrivals to the designated models. Developing a generic modelling-driver interface, would allow the integration of existing simulators with VEX. We are also implementing a modelling-driver interface, which would allow the integration of existing simulators with VEX. We demonstrated our methodology with a case study, where our prototype implementation of the methodology in Java, is used in conjunction with performance testing tools throughout the software lifecycle. The overall prediction accuracy provides evidence that our approach can make a useful contribution to current performance engineering practices.

REFERENCES