Validating Model-Driven Performance Predictions on Random Software Systems

Vlastimil Babka[1], Petr Tůma[1], and Lubomír Bulej[1,2]

[1] Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, 118 00 Prague, Czech Republic
{vlastimil.babka,petr.tuma,lubomir.bulej}@d3s.mff.cuni.cz
[2] Institute of Computer Science, Academy of Sciences of the Czech Republic
Pod Vodárenskou věží 2, 182 07 Prague, Czech Republic

Abstract. Software performance prediction methods are typically validated by taking an appropriate software system, performing both performance predictions and performance measurements for that system, and comparing the results. The validation includes manual actions, which makes it feasible only for a small number of systems.

To significantly increase the number of systems on which software performance prediction methods can be validated, and thus improve the validation, we propose an approach where the systems are generated together with their models and the validation runs without manual intervention. The approach is described in detail and initial results demonstrating both its benefits and its issues are presented.

Keywords: performance modeling, performance validation, MDD.

1 Motivation

State of the art in model-driven software performance prediction builds on three related factors: the availability of architectural and behavioral *software models*, the ability to solve *performance models*, and the ability to *transform* the former models into the latter. This is illustrated for example by the survey of model-driven software performance prediction [3], which points out that the typical approach is to use UML diagrams for specifying both the architecture and the behavior of the software system, and to transform these diagrams into performance models based on queueing networks.

Both the models and the methods involved in the prediction process necessarily include simplifying assumptions that help abstract away from some of the complexities of the modeled system, e.g., approximating real operation times with probability distributions or assuming independence of operation times. These simplifications are necessary to make the entire prediction process tractable, but the complexity of the modeled system usually makes it impossible to say how the simplifications influence the prediction precision.

G.T. Heinemann, J. Kofron, and F. Plasil (Eds.): QoSA 2010, LNCS 6093, pp. 3–19, 2010.

Without sufficient insight into the modeled system, a straightforward approach to the question of prediction precision would be similar to common statistical validation: a sufficiently representative set of systems would be both modeled and measured and the measurements would be compared with the predictions. Unfortunately, the fact that the software models still require manual construction limits the ability to validate on a sufficiently representative set of systems. For most prediction methods, the validation is therefore limited to a small number of manually constructed case studies.

To improve the validation process, we propose to automatically generate software systems together with their models and then use the systems for measurement and the models for prediction. That way, we can validate the performance predictions on a large number of systems, and, provided that the generated systems are representative enough, achieve a relatively robust validation of the prediction methods.

The goal of this paper is to explore the potential of the proposed validation process by taking a specific generation mechanism and applying it on a specific performance model. While neither the generation mechanism nor the performance model are the focus of the paper, we describe them in detail so that the reader can form an impression of what the benefits and the issues of applying the approach are. We start by analyzing the requirements on the automatic software system generation in Section 2 and detailing a particular generation mechanism in Section 3. In Section 4, we describe how we transform the architecture model of the generated software system into a performance model based on the Queueing Petri Nets (QPN) formalism [4]. Section 5 presents the initial results of the validation using the generation mechanism and the performance model outlined in the previous two sections. In Section 6, we discuss the broader context of the related work. We conclude by outlining future development directions in Section 7.

2 Requirements on Software Generation

While an automated generation of a fully functional software system would generally be considered infeasible, it is done in limited contexts, for example when generating domain specific editors [9], test cases [1,8], test beds [13], or benchmarks [33]. When generating software systems to validate performance predictions, the context is roughly as follows:

Executability. The system must be immediately executable, so that measurements of its performance can be taken.

Functionality. The system does not have to deliver any particular functionality, because the validation is concerned with other than functional properties.

Applicability. The performance effects observed on the generated system must be applicable to real systems for which the validated performance prediction method would be considered.

These requirements are commonplace when constructing benchmark applications for performance evaluation [28,24]. While not expected to deliver real results, a

benchmark is required to exercise the system in a manner that is as close to real application as possible, so that the benchmark performance is directly related to the performance of the corresponding production system.

When our goal is to see how the simplifying assumptions in the performance models influence the prediction precision, we can extrapolate the requirements easily: the generated system needs to be faithful to reality especially in those places where the prediction makes the simplifications. We now list some of the significant simplifications that are commonly performed:

Scheduling approximation. The operating system scheduler is usually a complex module that schedules threads or processes based not only on priorities or deadlines, but also on heuristics based on resource consumption or interactive behavior. Performance models usually simplify scheduling using one of the well defined algorithmic models such as random, first-come first-served, or ideal sharing [11].

Operation duration approximation. Although the individual operation durations tend to be arbitrary, performance models frequently approximate them with statistical distributions, especially the exponential distribution, which maps very well to some of the underlying analytical models. Only sometimes, arbitrary distributions are approximated [7].

Operation dependency approximation. The objects or functions of a software system frequently operate on shared data. This gives rise to a multitude of dependencies that influence performance. Perhaps most notable are dependencies due to locking, which are usually captured by performance models, for example as special tasks in LQN [30] or special places in QPN [4]. Less frequently captured are dependencies due to argument passing, so far only captured in performance models solved through simulation [19]. Otherwise, operation durations are usually assumed statistically independent.

Resource isolation approximation. Sharing of processor execution units, address translation caches, external processor buses and similar features found in contemporary hardware architectures potentially impacts performance. These are only rarely captured by performance models [10], and usually only for evaluating hardware design rather than software system. Typical performance models assume resources are mostly independent.

Although other approximations could likely be found, we focus on the ones listed here. Where the individual approximations are not concerned, we strive to make the generated software system random, to avoid introducing any systematic error into the validation. To provide a practical anchor, we apply our approach in the context of the Q-ImPrESS project [27], which deals with quality-of-service predictions in service oriented systems.

3 Generating Tree Structured Servers

In our experiments, the overall architecture of the generated software system has been influenced by the architectural model of the Q-ImPrESS project [5].

This model is built around the notion of components, whose interfaces are interconnected in a hierarchical manner, and whose behavior is described using the usual notions of operation invocations, sequences, branches and loops.

The generated software system consists of leaf modules, which approximate the primitive components of the architectural model, and interconnecting modules, which approximate the hierarchical interconnections and the behavior description of the architectural model. The leaf modules perform useful work in the sense of generating processing workload. The interconnecting modules arrange other modules in sequences, branches and loops.

In the following, we need to make a distinction between multiple meanings of some terms. We therefore use *module* when we mean a code unit implementing a feature, *component* when we mean an architectural element, and *instance* when we mean an actual state allocated at runtime. The same module can be used to realize multiple components. Multiple instances of the same component can be allocated.

We have opted to generate the system in top-to-bottom order. First, a module realizing the topmost component is selected at random from the set of all existing modules. Then, for any component realized by an interconnecting module with unassigned children, the same random selection is applied recursively to assign the child components. The probability of selecting a particular module is adjusted so that only interconnecting modules are selected at the topmost level of the architecture. Leaf modules are gradually more likely to be selected on the lower levels of the architecture.

The described algorithm generates a tree architecture. This would present an unacceptable restriction since there are no shared components in a tree architecture, and sharing of a component typically influences performance, especially when synchronization is involved. However, our design involves some synchronization between threads, explained later on. In principle, this synchronization resembles synchronization over a shared component, thus making the restrictions of the tree architecture relatively less important.

3.1 Realistic Leaf Modules

In the architecture, the workload that exercises the system is only generated by the leaf modules – the workload generated by the interconnecting modules amounts to the usual invocation overhead and remains trivial by comparison. Since we require the workload to exercise the system in a realistic manner, we use benchmarks from the SPEC CPU2006 benchmarking suite [14], which has been designed to reflect realistic workloads, as the leaf modules.

The use of benchmarks from the SPEC CPU2006 benchmarking suite brings multiple technical challenges related to reuse of code that was not designed to be strictly modular. To begin with, we have manually separated the initialization and execution phases of the benchmarks as much as possible, and wrapped each benchmark in a class with a unified module interface. The interface makes it possible to initialize all benchmarks before commencing measurement, and to execute each benchmark through a single method invocation.

Wrapping the benchmarks in classes is further complicated by the fact that the benchmarks are single threaded and use statically allocated data structures. Creating multiple instances of a wrapper class or invoking methods of a wrapper instance from multiple threads could therefore lead to race conditions. A straightforward solution, namely converting the statically allocated data into dynamically allocated attributes of the corresponding wrapper class, would require further modifications of the benchmarks. Given the size of the benchmarks, performing such modifications manually would not be a feasible approach – the potential for introducing subtle errors into the benchmarks that would disrupt the measurement is simply too high.

To tackle the problem, we have modified the linking process so that every component realized by a module uses a separate copy of the statically allocated data structures. In detail, for every component realized by a module, new copies of the binary object files containing the module and the corresponding wrapper class are created. In these copies, all external symbols are uniquely renamed. The generated software system is then linked with the renamed copies. This ensures that even when the generated software system contains multiple components initially realized by the same module, the individual components will be served by distinct module copies with distinct statically allocated data.

To cover the situation where methods of a wrapper class instance are invoked from multiple threads, each wrapper class is protected by a static lock. Thanks to symbol renaming, this lock synchronizes concurrent accesses to each single component, but not concurrent accesses to multiple components realized by the same module. As an exception, the lock is not used for modules whose correct function without synchronization can be reasonably expected, such as modules written by ourselves or modules explicitly documented as thread safe.

Since the standard execution time of the individual benchmarks can be in the order of hours, and the generated software system can incorporate many benchmarks arranged in sequences, branches and loops, the execution time of the entire system could become rather long. We have therefore also reduced the input data of the benchmarks to achieve reasonable execution times, and excluded some benchmarks whose execution times would remain too long even with reduced input data.

Besides the leaf modules that wrap the benchmarks from the SPEC CPU2006 suite, we have also included leaf modules that exercise the memory subsystem in a well-known manner. These leaf modules, together with the workload that they generate, are described in [2].

Whenever a module accepts arguments, the generated software system provides them as random values from a configurable statistical distribution or value domain. This goes not only for the arguments of the leaf modules, which include for example allocated memory block sizes, but also for the arguments of the interconnecting modules, which include for example loop iteration counts or branch path probabilities.

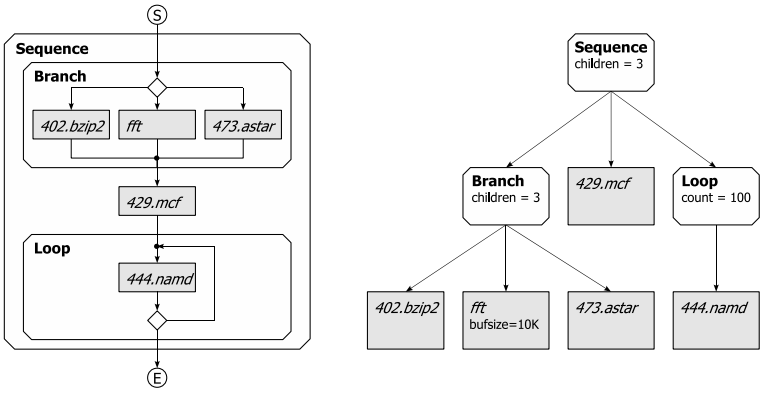

Fig. 1. Example of control flow and architecture of a generated system

3.2 Client Workload Generation

In the adopted architectural model of the Q-ImPrESS project [5], the topmost components represent services used by external clients. In this sense, each generated software system therefore also approximates a service used by external clients – and we need to provide the system with a mechanism for an appropriate client workload generation.

Our tool uses a closed workload model, with client requests handled by a thread pool. Each client request is handled by one thread – the thread invokes the topmost component of the generated software system, which, depending on the particular interconnecting module used, invokes some or all of its child components, progressing recursively through the entire system. The number of clients, the client think time, and the thread pool size are again random values with configurable properties.

To avoid excessive synchronization, threads allocate private instances of components. Coupled with the process of module wrapping and module linking, this makes it possible to execute multiple threads in parallel even in a leaf module that is single threaded by design, provided the module was selected to realize multiple components.

To summarize, the generated software system is a random tree of components that generate realistic workloads, executed in parallel by multiple threads with some degree of synchronization. An example of such a system is depicted on Fig. 1. This meets most of the requirements of Section 2, except for the requirement of faithful operation dependency approximation, which is still simplified because the components do not pass arguments among themselves.

4 Constructing Performance Models

The tool outlined in the previous sections provides us with a way to quickly generate a large number of executable systems together with their architectural

models. The next step is transforming the generated architectural models into performance models, and populating the performance models with input values such as operation durations and loop iteration counts. We perform this step in the context of performance models based on the QPN formalism [4].

4.1 Performance Model Structure

The process of constructing a performance model of a component system using the QPN formalism is outlined in [17]. To summarize, queueing places are used to model both client think time and server processing activity, immediate transitions describe the behavior of the component system, token colors are used to distinguish executing operations. Since our approach (implemented as an automated model-to-model transformation) is based on the same principle, we limit the description to significant aspects of the transformation.

The part of the network that is concerned with clients and threads uses a *clients* queueing place to model the client think time and a *threads* ordinary place to model the thread pool size. The tokens in the *clients* place represent thinking clients. The place has an infinite server scheduling strategy with exponential service time representing the client think time, the initial token population is equal to the number of clients. The tokens in the *threads* place represent available threads, with initial population equal to the size of the thread pool.

When a client finishes thinking and a thread for serving it is available, a transition removes one token from the *clients* place and one token from the *threads* place, and puts one token representing the thread serving the client into an ordinary place called *requests*.

The component operations are either active, actually consuming processor time, or passive, waiting without consuming processor time.

To model the execution of active operations, we use a queueing place called *processing*, configured to use the first-come first-served strategy, with the number of servers set to the number of processors available to the system. For each component whose operation is active, a dedicated color is defined, and the service time for that color in the *processing* place is set to reflect the operation duration.

In a similar manner, we use a queueing place called *waiting* to model the execution of passive operations. For each component whose operation is passive, a dedicated color is defined, and the service time for that color in the *waiting* place is set to reflect the waiting duration.

To model components that serialize execution, we use an ordinary place called *mutex*. We define a unique color for each serializing component operation and initially place one token of that color into the *mutex* place. The transition through which the tokens representing synchronized component operation arrive at the *processing* place additionaly removes the corresponding token from the *mutex* place ; analogously for returning the token.

The transitions around the *processing* and *waiting* places capture the flow of control through the individual components as defined by the architectural model. The transformation traverses the architecture model in a depth-first order and defines transitions that reflect the sequence of component operations that a

thread serving a client request will perform. Following are the major rules for defining the transitions:

- Each component operation connects to the previous operation by removing the token from where the previous operation deposited it, the first such transition removes the token from the *request* place.
- Each active operation is modeled by a transition that connects to the previous operation and deposits a token of the color corresponding to this operation into the *processing* place.
- Each passive operation is modeled by a transition that connects to the previous operation and deposits a token of the color corresponding to this operation into the *waiting* place.

A component that realizes a branch needs two additional ordinary places, called *split* and *join*. One transition connects the previous operation to the *split* place with a unique color and mode for each child, all firing weights are set equally. Additional transitions to the *join* place connect the child component operations.

A component that realizes a loop needs one additional ordinary place, called *loop*, and two colors, called *looping* and *exiting*, used in the *loop* place to indicate whether the loop will execute the next iteration. Two transitions deposit a token in the *loop* place, one connects the previous operation, one returns the tokens of the finished child component operations. Both transitions leading to the *loop* place have two modes that deposit a token of either the *looping* or the *exiting* color. The number of times a token of the *looping* color is deposited into the *loop* place has a geometric distribution, the firing weights are calculated so that the mean value of the distribution is equal to the loop iteration count.

4.2 Providing Operation Durations

The performance models need to be populated with input values. There are two kinds of input values – values such as loop iteration counts or branch path probabilities, which are available as the arguments of the interconnecting modules, and are therefore obtained directly from the generated software system – and values such as operation durations, which have to be measured.

To measure the generated software system, we employ the advantage of having full control over code generation, and insert the necessary instrumentation directly into the wrapper classes of the individual modules. The executable system therefore reports its own performance, providing us with both the input values of the performance model and the overall performance figures to be compared with the outputs of the performance model.

The instrumentation includes the necessary precautions to avoid systematic measurement errors – the component initialization, buffer allocation, and warmup execution take place before the measurements are collected, reporting takes place after the measurements terminate. Operation durations are measured twice, once for each component executing in isolation and once for each component executing together with the rest of the generated software system. Both the wall clock times and the thread local times are recorded.

5 Validation Results

With both the generated executable systems and the corresponding performance models at hand, we carry out the validation by comparing the outputs of the performance models with the measurements of the executable systems. Before the results of the validation are presented, however, we point out that our goal is not to evaluate the prediction precision of a specific performance model, but rather to explore the potential of the proposed validation process. The result presentation is structured accordingly.

Our measurements were taken on a Dell PowerEdge 1955 system.[1] The executable systems used a thread pool of eight threads. The performance models were solved using the SimQPN solver [18].

Unless specified otherwise, we report and compare client service time, measured from the moment the client request is assigned a thread to the moment the client request is finished.

5.1 Precise Single Client Predictions

As an example of a context where precise performance predictions are expected, we have first generated 228 software systems and configured them for a single client workload, thus minimizing resource contention due to parallel execution. The results are shown on Fig. 2 as a ratio of predicted to measured service times plotted against the predicted processor utilization.

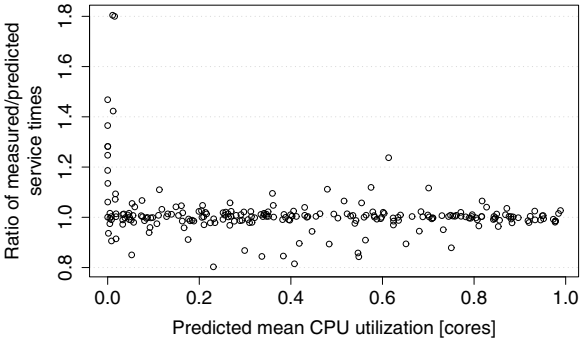

Fig. 2. Prediction precision with one client

We can see that the prediction error is relatively small, except for the errors at very low processor utilization, where the service time is very small compared to the client think time, and the error can therefore be attributed to the overhead of the execution infrastructure. In the following, we avoid this error by selecting

[1] Dual Quad-Core Intel Xeon processors (Type E5345, Family 6, Model 15, Stepping 11, Clock 2.33 GHz), 8 GB Hynix FBD DDR2-667 memory, Intel 5000P memory controller, Fedora Linux 8, gcc-4.1.2-33.x86_64, glibc-2.7-2.x86_64.

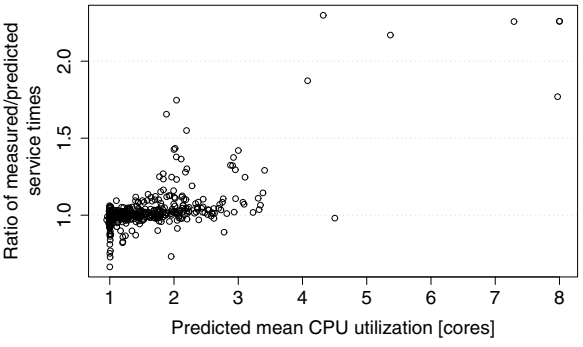

Fig. 3. Prediction precision with multiple clients

only those results with predicted processor utilization for a single client workload above 2 %. On Fig. 2, this would leave 204 remaining architectures, of which 8 % exhibits prediction error over 10 %.

5.2 Multiple Clients Prediction

We move on to a context with multiple clients to assess the prediction error when multiple concurrent threads are involved. For this experiment, we have used the same architectures as in Section 5.1, with the number of clients set to 50. To increase the coverage, additional 600 architectures were generated, with the number of clients drawn from a uniform distribution between 5 and 30, and then restricted to cases where the intensity of requests generated by the clients was predicted to result in mean thread utilization of at least four of the eight available threads. After the filtering described in Section 5.1, 645 architectures remained.

The results are shown on Fig. 3, where the ratio of predicted to measured service times is again plotted against the predicted processor utilization. We can see that the prediction accuracy is lower than in the single client scenario, and that the error tends to increase with growing processor utilization.

5.3 Operation Duration Distribution

Using the same architectures as in Section 5.2, we examine how the approximation of the individual operation durations with statistical distributions impacts the prediction precision. The SimQPN solver uses discrete event simulation and therefore easily supports multiple statistical distributions – for illustration, we compare the results from models that use normal distribution with the results from models that use exponential distribution. For normal distribution, the parameters were set to match the sample mean and variance of the isolated operation durations. For exponential distribution, the mean was set to match the sample mean of the isolated operation durations.

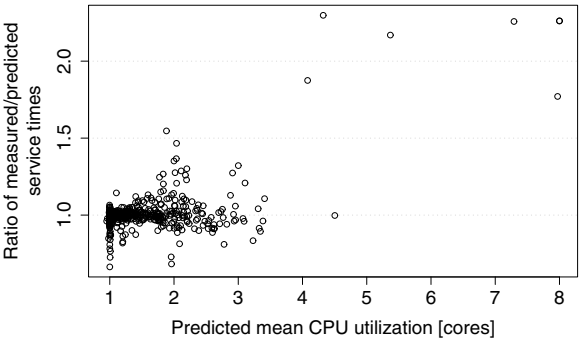

Fig. 4. Prediction precision with multiple clients, using exponential approximation of isolated durations for the service time prediction

The results that use exponential distribution are shown on Fig. 4. By comparing them with the results that use normal distribution on Fig. 3, we can observe that in many cases, the ratio of predicted to measured service times is lower for the exponential distribution than the normal distribution. This can occur both for the optimistic predictions, where the error is thus decreased, and for the accurate predictions, where the error turns them into pessimistic ones.

To explain this effect, we turn to Fig. 5, which shows the ratio between the results for the two distributions, plotted against the mean number of threads that are blocked due to the serializing modules. We can see that in the extreme cases (when there is no blocking, or when a serializing component dominates the execution so much that seven out of eight threads are blocked), both distributions yield the same prediction. When a mix between serialized and concurrent execution is present, using the exponential distribution can yield a significantly more pessimistic prediction than using the normal distribution.

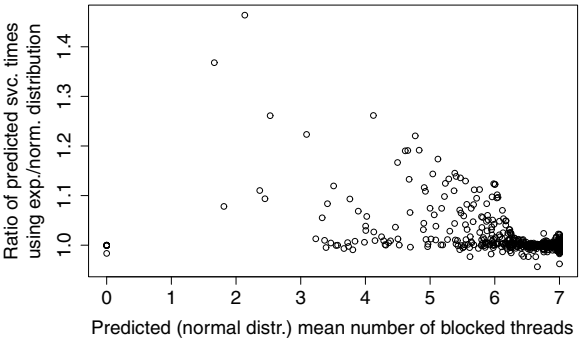

Fig. 5. Difference in prediction with exponential and normal approximation of isolated durations, depending on the mean number of threads blocked due to synchronization

A mix between serialized and concurrent execution is present when a serializing component is close to dominating the execution and threads are starting to block. Due to variations in operation durations, invocations can arrive at the serializing component either sooner or later than usual – but while the invocations that arrive sooner are more likely to be queued, the invocations that arrive later can more likely be processed immediately. In effect, this amplifies variations that lead to pessimistic predictions, because the performance gain due to invocations arriving sooner is masked by queueing, while the performance loss due to invocations arriving later is not. In our experiment, we have chosen distribution parameters that match the isolated operation durations, which resulted in the exponential distribution having higher variance than the normal distribution. The amplifying effect is thus more pronounced with the exponential distribution.

5.4 Resource Contention Impact

Many performance prediction methods involve performance model calibration, where the operation durations are adjusted to fit the model output onto the measurement results. When the operation durations are influenced by resource contention that is not captured in the performance model, such as contention for various memory caches, the calibration incorporates the resource contention effects into the adjusted operation durations. We isolate this effect by populating the performance models with the operation durations measured when the components were executing together, as opposed to the operation durations measured when each component was executing in isolation, used so far. Thread local times are used to obtain the new durations, thus excluding waiting on serialization locks, which is already modeled explicitly in the performance model.

The results where performance model is populated by isolated measurements were shown on Fig. 3, the results of model populated with measurements from combined execution are shown on Fig. 6. We observe that using measurements from the combined execution eliminates most of the cases where the predicted

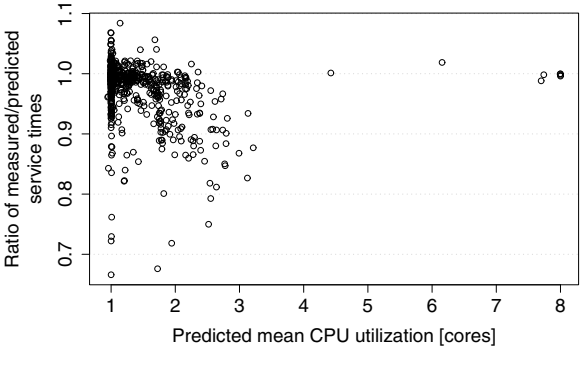

Fig. 6. Prediction precision with multiple clients, using measurements from combined execution to parameterize the performance model

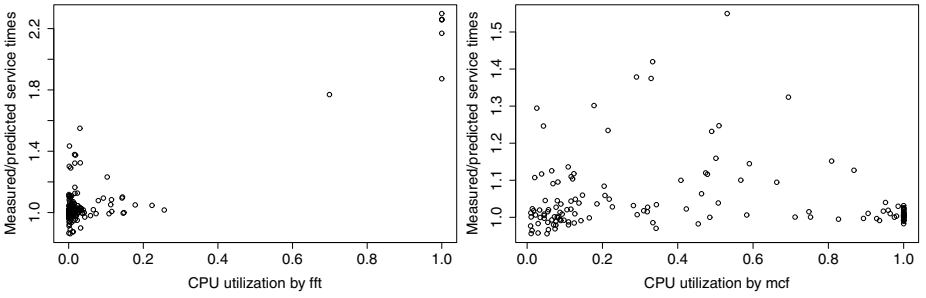

Fig. 7. Prediction error depending on the relative CPU utilization of a module

service times were lower than the actual measured times. Since resource contention depends on the particular choice of components, we also determine how the individual components contribute to this effect. To do this, we plot the prediction error with the operation durations measured in isolation against the relative processor utilization due to a particular component. To conserve space, we only show plots for the `fft` and `mcf` modules, on Fig. 7.

The `fft` module does not serialize execution and is known to be sensitive to cache sharing effects – as expected, the prediction error increases as the `fft` workload dominates the execution time. The `mcf` module serializes execution – when the `mcf` workload usurps the execution time, the serialization limits the competition for resources and the prediction precision is good. In contrast, when the `mcf` workload consumes around half of the execution time, the prediction error is significant for some systems, most likely when the components consuming the other half of the execution time compete with `mcf` for resources.

5.5 Workload Scalability Behavior

Often, the goal of performance prediction is not to predict throughput or response time with a fixed number of clients, but to determine scalability of the system depending on the number of clients. Our approach can be used to validate such performance prediction in a straightforward way, by running both the generated executable and prediction with varied number of clients.

Figure 8 shows an example of such validation for one of the systems which yielded imprecise prediction in Section 5.2. We can observe that the throughput predicted using normal distribution of operation durations is accurate up to six clients, then the error starts to increase. The prediction using exponential distribution of operation durations has a similar trend, but with relatively lower values, pessimistic up to nine clients and optimistic afterwards.

Finally, note that the validation results might be somewhat deceptive because the frequency of occurence of particular system properties might be different between generated and real systems.

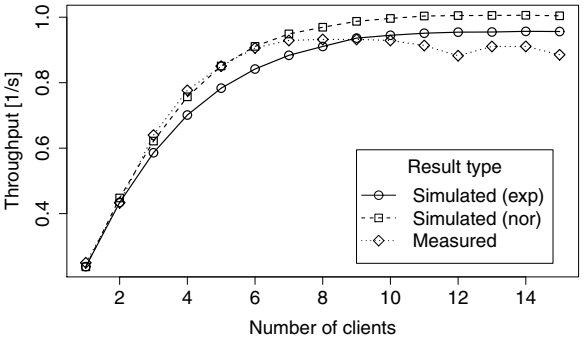

Fig. 8. Predicted and measured throughput depending on the number of clients

6 Related Work

Most works that focus on a particular performance prediction methodology strive to validate it on a case study that is reasonably large and undisputedly realistic. Recent examples of such validation include using the SPEC jAppServer 2004 workload running on WebSphere and Oracle [17], using an online stock trating simulation with two different architectures and two different platforms [21,20], using a generic performance model of an EJB server in various scenarios [32], or using a model of an air traffic control system [12]. While the cited validation examples are without doubt more realistic than our approach, they are also a case in point we are making: the expenses of manually constructing the performance models prevent validation on a large number of case studies.

The situation is somewhat alleviated by the existence of case studies designed to provide a playground for validation activities – besides application benchmarks such as SPEC jAppServer [25], Sun Pet Store [26], or RUBiS [22], we can list the CoCoME competition [23], which provides both the architecture model and the prototype implementation. Even those case studies, however, do not amount to the hundreds of systems which we have used here.

Besides validating a particular methodology, the question of prediction precision is tackled from many other angles, including workload characterization [29,16], or generation of either the executable system [31,13,33,6] or the performance model [15,19]. What these approaches have in common is that they still require manual intervention, either in creating models of application architecture, performance, and workload, or in test execution. We eliminate the need for manual intervention by automatically generating both the executable software system and the corresponding performance model, rather than constructing one from the other.

A pivotal issue of our approach is the degree of realism achieved in the generated software systems. One of the ideas we have investigated to guarantee some degree of realism was to use common software metrics to assess the generated software systems and to exclude those with unusual values.

Interestingly, the common software metrics have turned out to be unsuitable. Some, such as the development effort or the function points, are not suitable for generated software systems with no specific functionality. Others, such as McCabe's cyclomatic number or Halstead's program measures, are focused on human perception of complexity, and thus tell little about whether the generated software system is realistic from the performance modeling perspective. Finally, measures such as lines-of-code count have usual values ranging across several orders of magnitude and therefore do not allow to reasonably exclude the generated software systems.

7 Conclusion

We have presented an approach that makes it possible to validate performance predictions on a large number of executable systems and performance models without excessive manual intervention. We believe the approach has the potential to bring a new type of answers to the important questions of performance modeling, such as "what is the range of prediction precision ?" or "what design patterns or design artifacts make performance difficult to predict ?" ... As a proof of concept, we illustrate the process of providing answers to some of these questions for performance models based on the QPN formalism (here, however, we focus more on the potential of the approach than the specific answers for this particular class of performance models).

Our approach has been implemented in a prototype tool that solves a number of technical issues, especially issues related to safe reuse of code that was not designed to be strictly modular. The prototype tool is available for download at http://d3s.mff.cuni.cz/benchmark.

The presented approach hinges on our ability to generate reasonably realistic systems. Although there might be some alleviating circumstances – for example, when the validation fails on a model that subsequent manual inspection finds unrealistic, we simply exclude the model – the question of realism remains too important to be ignored. In this work, the elements contributing to the overall realism are for example the usage of a real server model and a real workload, but other elements, for example the data dependencies, remain to be tackled.

In the absence of a clearly objective measure or criteria of realism, it might turn out that only more extensive usage of the approach will provide enough practical knowledge to answer the issue of realism to satisfaction.

Acknowledgments

The authors gratefully acknowledge Samuel Kounev for his help with SimQPN and Jakub Melka and Michal Bečka for their work on the random software generator. This paper was partially supported by the Q-ImPrESS research project by the European Union under the ICT priority of the 7^{th} Research Framework Programme, by the Czech Science Foundation grant 201/09/H057, and by the grant SVV-2010-261312.

References

1. Avritzer, A., Weyuker, E.J.: The Automatic Generation of Load Test Suites and the Assessment of the Resulting Software. IEEE Trans. Software Eng. 21(9) (1995)
2. Babka, V., Bulej, L., Decky, M., Kraft, J., Libic, P., Marek, L., Seceleanu, C., Tuma, P.: Resource Usage Modeling, Q-ImPrESS Project Deliverable D3.3 (2008), http://www.q-impress.eu/
3. Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M.: Model-Based Performance Prediction in Software Development: A Survey. IEEE Trans. Software Eng. 30(5) (2004)
4. Bause, F.: Queueing Petri Nets - A Formalism for the Combined Qualitative and Quantitative Analysis of Systems. In: Proc. 5^{th} Intl. W. on Petri Nets and Performance Models. IEEE CS, Los Alamitos (1993)
5. Becker, S., Bulej, L., Bures, T., Hnetynka, P., Kapova, L., Kofron, J., Koziolek, H., Kraft, J., Mirandola, R., Stammel, J., Tamburrelli, G., Trifu, M.: Service Architecture Meta Model, Q-ImPrESS Deliverable D2.1 (2008), http://www.q-impress.eu/
6. Becker, S., Dencker, T., Happe, J.: Model-driven generation of performance prototypes. In: Kounev, S., Gorton, I., Sachs, K. (eds.) SIPEW 2008. LNCS, vol. 5119, pp. 79–98. Springer, Heidelberg (2008)
7. Becker, S., Koziolek, H., Reussner, R.: The Palladio Component Model for Model-driven Performance Prediction. J. Syst. Softw. 82(1) (2009)
8. Bertolino, A.: Software Testing Research: Achievements, Challenges, Dreams. In: Proc. Intl. Conf. on Software Engineering, ICSE 2007, W. on the Future of Software Engineering, FOSE 2007. IEEE CS, Los Alamitos (2007)
9. Budinsky, F., Brodsky, S.A., Merks, E.: Eclipse Modeling Framework. Pearson Education, London (2003)
10. Cascaval, C., DeRose, L., Padua, D.A., Reed, D.A.: Compile-Time Based Performance Prediction. In: Carter, L., Ferrante, J. (eds.) LCPC 1999. LNCS, vol. 1863, p. 365. Springer, Heidelberg (2000)
11. Franks, G., Maly, P., Woodside, M., Petriu, D.C., Hubbard, A.: Layered Queueing Network Solver and Simulator User Manual (2005), http://www.sce.carleton.ca/rads/lqns/
12. Franks, G., Al-Omari, T., Woodside, M., Das, O., Derisavi, S.: Enhanced Modeling and Solution of Layered Queueing Networks. IEEE Trans. Software Eng. 35(2) (2009)
13. Grundy, J.C., Cai, Y., Liu, A.: Generation of Distributed System Test-Beds from High-Level Software Architecture Descriptions. In: Proc. 16^{th} IEEE Intl. Conf. on Automated Software Engineering, ASE 2001. IEEE CS, Los Alamitos (2001)
14. Henning, J.L.: SPEC CPU2006 Benchmark Descriptions. SIGARCH Comput. Archit. News 34(4) (2006)
15. Hrischuk, C.E., Rolia, J.A., Woodside, C.M.: Automatic Generation of a Software Performance Model Using an Object-Oriented Prototype. In: Proc. 3^{rd} Intl. W. on Modeling, Analysis, and Simulation On Computer and Telecommunication Systems, MASCOTS 1995. IEEE CS, Los Alamitos (1995)
16. Joshi, A., Eeckhout, L., Bell Jr., R.H., John, L.K.: Distilling the Essence of Proprietary Workloads Into Miniature Benchmarks. ACM Trans. Archit. Code Optim. 5(2) (2008)
17. Kounev, S.: Performance Modeling and Evaluation of Distributed Component-Based Systems Using Queueing Petri Nets. IEEE Trans. Software Eng. 32(7) (2006)

18. Kounev, S., Buchmann, A.: SimQPN: A Tool and Methodology for Analyzing Queueing Petri Net Models by Means of Simulation. Perform. Eval. 63(4) (2006)
19. Koziolek, H., Happe, J., Becker, S.: Parameter dependent performance specifications of software components. In: Hofmeister, C., Crnković, I., Reussner, R. (eds.) QoSA 2006. LNCS, vol. 4214, pp. 163–179. Springer, Heidelberg (2006)
20. Liu, Y., Fekete, A., Gorton, I.: Design-Level Performance Prediction of Component-Based Applications. IEEE Trans. Software Eng. 31(11) (2005)
21. Liu, Y., Gorton, I.: Accuracy of Performance Prediction for EJB Applications: A Statistical Analysis. In: Gschwind, T., Mascolo, C. (eds.) SEM 2004. LNCS, vol. 3437, pp. 185–198. Springer, Heidelberg (2005)
22. OW2 Consortium: RUBiS: Rice University Bidding System, `http://rubis.ow2.org/`
23. Rausch, A., Reussner, R., Mirandola, R., Plasil, F. (eds.): The Common Component Modeling Example: Comparing Software Component Models. Springer, Heidelberg (2008)
24. Standard Performance Evaluation Corporation: SPEC CPU2006 Benchmark, `http://www.spec.org/cpu2006/`
25. Standard Performance Evaluation Corporation: SPECjAppServer2004 Benchmark, `http://www.spec.org/jAppServer2004/`
26. Sun Microsystems, Inc.: Java Pet Store Demo, `http://blueprints.dev.java.net/petstore/index.html`
27. The Q-ImPrESS Project Consortium: Quality Impact Prediction for Evolving Service-oriented Software, `http://www.q-impress.eu/`
28. Transaction Processing Performance Council: TPC Benchmarks, `http://www.tpc.org/information/benchmarks.asp`
29. Weyuker, E.J., Vokolos, F.I.: Experience with Performance Testing of Software Systems: Issues, an Approach, and Case Study. IEEE Trans. Software Eng. 26(12) (2000)
30. Woodside, C.M., Neron, E., Ho, E.D.S., Mondoux, B.: An "Active Server" Model for the Performance of Parallel Programs Written Using Rendezvous. J. Syst. Softw. 6(1-2) (1986)
31. Woodside, C.M., Schramm, C.: Scalability and Performance Experiments Using Synthetic Distributed Server Systems. Distributed Systems Engineering 3(1) (1996)
32. Xu, J., Oufimtsev, A., Woodside, M., Murphy, L.: Performance Modeling and Prediction of Enterprise JavaBeans with Layered Queuing Network Templates. SIGSOFT Softw. Eng. Notes 31(2) (2006)
33. Zhu, L., Gorton, I., Liu, Y., Bui, N.B.: Model Driven Benchmark Generation for Web Services. In: Proc. 2006 Intl. W. on Service-oriented Software Engineering, SOSE 2006. ACM, New York (2006)