Architecture-Based Performance Simulation

Gero Decker, Volker Gersabeck, Jan Schaffner and Marcel Seelig

Abstract—When developing large software systems it is hard to predict their performance behavior at early development stages. This paper presents an engineering methodology for performance prediction based on high-level application architectures that are simulated within the real infrastructure setup. This methodology is realized in a performance simulation framework. Our results show that a software system's performance can be predicted based on its conceived high-level software architecture already at early phases in the software development process. Then the approach was validated with a case study, where the performance of an existing system was measured. Accordingly we created a performance model, and ran simulations within our framework.

Index Terms—performance case studies, performance evaluation, performance-oriented design, simulation techniques and experimental design, performance tools

I. INTRODUCTION

Performance is a key aspect for software systems. Bad responsiveness can result in a decreasing number of users. In contrast, exceptional performance can lead to a decisive competitive advantage, as the example of Google's search engine shows.

However, performance issues of enterprise applications are often poorly addressed during the software development process. In industry practice, performance issues are tackled with both “tune for performance”- and “design for performance”-approaches. When following a “tune for performance” strategy, important architectural design choices are normally made without having performance in mind. Later, after having fully implemented the system, hot spots are identified and fixed. Problematically, the most critical performance problems are often due to wrong architectural decisions in early stages of the development process. Therefore, a “tune for performance” development strategy will often result in major reimplementation work and is thus ineffective. Performance concerns, as stated by Jain [1] are influencing every step of the software system life cycle: requirements specification, design, development, implementation and manufacturing, sales and purchase, use and upgrade. Still it is a highly argued question when to think and act to get a reasonable performance. No one would deny that performance is important for every software project. Yet it is often considered less important than other factors and moved to a later part of the software’s life cycle as documented by Dugan [2]. This decision is often justified with Knuth’s famous statement:

We should forget about small efficiencies, about 97% of the time. Premature optimization is the root of all evil[3].

In order to avoid unnecessary reimplementation work, “design for performance” tries to incorporate performance trade-offs into early architecture definition stages. Architectural patterns help to avoid common pitfalls.

Using performance prediction techniques facilitates more accurate “design for performance” than simple rules of thumb. Here, we can distinguish between two approaches based on different types of performance models: Analytical performance models and simulation models. Analytical models such as Queuing Systems explained by Kleinrock [4] are based on a set of formulas that can be used to generate performance metrics for given parameters. Simulation models are software systems that simulate the performance behavior of the real system.

We assume that the biggest part of a modern enterprise system is already available at very early development stages. Existing components (or services) are incorporated into the system and a lot of the actual work is done by the sophisticated middleware platforms. Therefore, the middleware must not be neglected when predicting a software system’s performance. We argue that analytical performance models are not applicable in the industry, which is due to the fact that the respective mathematical models are very hard to create. In contrast, the complexity of modern middleware platforms is huge and can hardly be captured using mathematical models.

Based on these assumptions our simulation models go one step further than what is described e.g. in the workings of Denaro et al. and Liu et al. [5] [6]: We propose a method for analysis of a software system’s performance based on its conceived architecture. In our approach, we try to use as much of the real environment as possible. We argue that much better performance predictions can be done that way while having little modeling effort for the engineers. We have implemented our concepts in a simulation framework. Our results show that a software system’s performance can be predicted already at very early stages in the design phase of the software development process, if architecture-based performance models are used to carry out simulations within real middleware infrastructure.

We validate our approach using a case study. We have taken an existing ABAP system, measured the performance of the system, and compared the measurements to simulation results obtained with our framework.

The remainder of this paper is structured as follows. First, we discuss related work in section II. In section III we introduce our methodology for simulation-based performance engineering. Section IV discusses the implementation of our performance simulation framework. Afterwards, section V
presents the case study where our simulation-based approach has been applied. The results demonstrating the applicability of our approach are discussed in section VI, also providing concluding remarks and an overview of our further research prospects.

II. RELATED WORK

The assessment of software performance through simulation is certainly a very interesting and important topic, but very difficult to solve in a generic way - which is perhaps why not too much work in this area exists. A survey by Balsamo et al. [7] presents performance models to characterize the quantitative behavior of software systems. It also concludes that, despite successful application of the different models, performance prediction has not been integrated into normal software development. A comprehensive review of recent research in the field of model-based performance prediction at software development time can be found in the aforementioned survey, whereas some representatives are also briefly discussed here. With all of these model-based approaches it is possible to predict an application's performance, each of them offering some specific advantages or disadvantages. They are applicable at different stages of the software lifecycle and have more or less maturity to be used for meaningful performance prediction. However, all of them require a lot of expertise in the area of performance, which is, according to an expert working for 28 years in the field of optimizing business applications, acquired only after many years of experience.

Microsoft’s magpie project[8], investigating performance of loosely coupled systems, develops a performance monitoring tool for Microsoft Windows which has the ability to account the resources consumed by a particular transaction throughout its lifetime. The performance application programming interface (PAPI) project[9] gives users a graphical representation of performance information that has been gathered by accessing hardware performance counters available on most modern microprocessors. Thus, it is allowing users to quickly see where performance bottlenecks are in their application. Although both projects are very useful for the refinement of workload models and localizing hot-spots, a fully implemented system is needed and therefore not really applicable during the early stages of software development.

Weyuker and Vokolos[10] discuss an approach to software performance testing using a case study. In their approach, a representative workload using history data is created and applied on an earlier version of the system to be tested, if available. They also found that those projects that are in the most trouble almost never considered performance issues during the architecture phase of development. They recognized that, with regard to performance, it was irrelevant what the software under test is actually doing, provided the resources were used in a similar way to the intended new software. This perception is very similar to our approach.

Early performance testing is investigated by Denaro et al.[5]. They state the need for early evaluation of software performance and found an existing focus on analytic models rather than testing techniques. Therefore they concentrate on performance testing applied to existing middleware products. For them, as for the authors of this work, early performance evaluation is desirable and existing software infrastructure, such as middleware, can be already used for performance prediction before any other component is developed. There are two major reasons for that assumption: First a great deal of resources is consumed on the middleware level and second many (business-) applications rely on the input of other applications or databases which is already available at the very beginning of a software project. Based upon those assumptions Denaro et. al. are generating stubs for missing components and are executing the test while applying workloads derived from use-cases. ForeSight [6] construct models based on empirical testing that act as predictors of the performance effects of architectural trade-offs. They use the results of an empirical benchmarking engine to build mathematical models to predict performance. In our approach a different path is chosen: architectural relevant components are represented by a generic business module, which can be configured to behave, with regard to performance, like the desired component.

Software Performance Engineering (SPE) as presented by Smith[11], is an approach to design software systems for performance. With SPE it is also possible to evaluate complex distributed enterprise applications. The performance models are basically analytical models that can be solved using formulas and input parameters to determine the system’s performance. That approach distinguishes SPE massively from our technique; although both are addressing performance at the same very early stage of software development with the emphasis on software architecture. With SPE a design for performance is possible albeit a complex infrastructure will easily make this task a difficult and time consuming one. Additionally this task has to be done by a performance expert.

Woodside, Petriu and Amyot, main investigators in the Performance from Unified Model Analysis (PUMA)[12] project, are developing a unified approach to building performance models from design models that specify scenarios for use cases. The performance modeling is done using the Layered Queuing (LQN) formalism. Their scenario input is based on the UML:SPT[13] profile. Layered Queuing Networks are used for component based performance prediction by Wu, McMullan, and Woodside [14]. The LQN notation was also proposed for future enhancement of simulation based performance testing by [5].

When applying these different approaches to the development of enterprise applications there is one central aspect that immediately attracts a software engineer’s attention: It costs a lot of time to come to an appropriate performance prediction. One reason for this dilemma is the fact that enterprise applications utilize technically very complex environments. Enterprise integration platforms belong to the biggest software systems in terms of lines of code. The performance behavior of this environment is hard to capture in analytical performance models. To our knowledge no research project has ever managed to do so with a satisfying precision. Another major disadvantage of model based performance prediction and analysis is that the infrastructure has to be modeled explicitly which takes a lot of time.
Simulation-based approaches are far more promising in this case. The next section describes our approach that aims to obtain performance predictions even when it comes to the use of complex infrastructure.

III. A METHODOLOGY FOR SIMULATION-BASED PERFORMANCE ENGINEERING

Throughout this section we describe our method for simulating software systems prior to their implementation, based on high-level architectural models. The five steps of the methodology are depicted in figure 1. Before we provide details for each step, we will introduce the modeling notation used for the performance models.

A. FMC Modeling Notation

The modeling approach is based on the Fundamental Modeling Concepts (FMC), an approach for describing architectural structures of computer based systems, using a semiformal graphical notation [15][16]. In order to support a wide variety of systems, FMC distinguishes three basic types of system structures which are fundamental aspects of any computer based system:

- Compositional structure, i.e. the static structure consisting of the interacting components of the system.
- Dynamic structure, i.e. the behavior of the components.
- Value structure, i.e. the data structures found in the system.

Only the compositional structures are relevant in the context of this paper and in consequence the corresponding conceptional and notational elements will be discussed below. Any system can be seen as a composition of collaborating components called agents. Each agent serves a well-defined purpose and communicates via channels with other agents. If an agent needs to keep information over time, he has access to at least one storage where information can be stored. Channels and storages are (virtual) locations where information can be observed. The agents are drawn as rectangular nodes, whereas locations are symbolized as rounded nodes. In particular, channels are depicted as small circles and storages are illustrated as larger circles or rounded nodes. Any agents and locations drawn with a shadow represent more than one exemplar of that type. The possibility to read information from or write information to a location is indicated by arrows. Types of agents and locations are identified by descriptive textual labels. Arbitrary complex structures can be described because agents can be connected to multiple locations and locations can be shared by multiple agents. For example, it is possible to describe unidirectional, bidirectional channels (connecting only two agents) as well as broadcast channels (connecting more than two agents) and channels for sending requests (bidirectional, with an "R"- arrow indicating the request direction). Shared storages can be used for buffered communication. In general, agents and locations are not necessarily related to the system’s physical structure. The compositional structure facilitates the understanding of a system, because one can imagine it as a physical structure (e.g. as a team of cooperating persons).

B. Iterative Methodology

Our approach focuses on the iterative process of simulating enterprise applications (i.e., the step-wise refinement of the software architecture based on the simulation results). A precondition for the iterative engineering cycles is that the following information is already available:

- The system’s most important use cases have to be known.
- The workflow intensity describes the work placed on the system by the clients. Metrics such as tasks performed per user per hour, or number of users can be applied for each use case.
- The performance goals define acceptable performance behavior per task. E.g. it could be demanded that searching for a certain item should not take longer than 0.01sec.

The whole engineering process is depicted in figure 2. The rounded rectangles represent artifacts / information while the other rectangles represent engineering activities. The arrows show which information is consumed by a certain activity and which information is produced.

As stated earlier in this section, the following five steps make up an iteration cycle:

1) Static Architecture Definition. The software engineer defines which building blocks will appear in the application and how they are connected. The system components might have already been developed, e.g. existing services that are being reused in the new system, while other system components have to be developed.

The static structure definition phase leads to the application’s static architecture. This is a very important artefact, because it is consumed by the two downstream activities “system landscape definition” and “dynamic architecture definition”.

2) System Landscape Definition. The static structure only gives a logical view of the system. During the system landscape definition the software engineer enriches the static architecture with system setup information. To do so, he describes how the system components are physically distributed (e.g., in terms of IP addresses) and defines on which technology platform the components will be realized (e.g., J2EE or ABAP). Channels are annotated with information about the protocol used by two agents for communication (e.g., HTTP or RMI). Configurations for storages indicate parameters such as the desired fill level of the database and empirical distribution of the data.

In order to keep the process of creating these models as simple as possible, the engineer has to be restricted in the choice of parameters. The final landscape definition has to correspond to the real system setup.

3) Dynamic Architecture Definition. The dynamic architecture describes the interactions between agents for a set of use cases. Request / response messages and the activities performed by the agents are characterized. As depicted in figure 2, the static architecture is necessary for describing the systems dynamic architecture. This is due to the fact that only those agents that are connected
by a channel in the static architecture can legally communicate via request / response messages. More details on the definition of the agents’ activities can be found in the section IV.

4) Simulation. Before a simulation can start the engineer has to select a set of use cases that are to participate in the simulation. The workload intensity will apply accordingly. Additionally, workload for other applications that are deployed in the target environment can be defined. That way the influence different applications have on each other, can be captured.

Simulations are divided into three phases: Configuration, execution and collection of results. During the configuration phase dummies are deployed to the machines and configured according to the system architecture. Storages are set up by the accessing dummies. One Generic Simulation Component (GSC) represents one agent from the static architecture. Since individual views on the dynamic architecture are propagated to the dummies, they know which calls they have to perform when they are triggered. After the configuration has finished the execution of the simulation can start. The system components are triggered according to the workload intensity and performance measurements are traced. As soon as the execution is over the trace data can be collected and aggregated into the simulation results.

5) Goals and Results Comparison. The results produced in the simulation give an insight into the conceived system’s performance behavior. As performance goals are set in the requirements, we can now compare them to the simulation results. This feedback can be used in two ways: On the one hand the architecture can be modified until the performance goals are met (e.g. by introducing caching mechanisms or by changing the physical setup). On the other hand the information on potential hot-spots can be used as input for the implementation phase in order to know where special care has to be taken.

IV. ARCHITECTURE OF THE PERFORMANCE SIMULATION FRAMEWORK

The purpose of this section is to describe the architecture and concepts of our performance simulation framework. This framework is an implementation of our methodology together with an editor for the models. The architecture of this framework is shown in figure 3.

The performance architect has access to the editor for creating the performance model. This model consists of the static architecture combined with the system landscape information and the dynamic architecture. Both the static and the dynamic architecture are defined using graphical representations of their models. The system landscape information is defined by parameters that can be added to the different parts of the models.

When the performance model is defined the simulation can be started. The simulation runner is responsible for the execution of the whole simulation. Using different adapters, GSCs (configurable generic business components) can be accessed. This enables the simulation to be independent of the technologies the GSCs are realized with. The current implementation of the framework supports J2EE and ABAP technology.
An adapter supports the three phases of simulation: configuration, execution, and result collection. At first the simulation runner distributes the different configurations to all GSCs through the adapters. Then the actual simulation will be started, during which all GSCs trace measurements. These measurements can be collected after the simulation has finished. The simulation runner aggregates the measurements and provides them as the simulation results to the result visualizer. This component uses the graphical representations of the performance model to visualize the results. The visualization uses textual annotations as well as colors to present the results to the performance engineer. Examples of both the performance model as well as the visualization are shown in section V.

A. Types Of GSCs

The current implementation of the framework supports three types of GSCs: J2EE dummies, ABAP dummies, and client dummies. The client GSC provides to the simulation runner the same interface as the adapters do. This way there is no difference for the simulation runner between an adapter and a client GSC. On the other hand, the client GSC does not need to forward any information to a GSC component, as it itself is the GSC component.

In general, each GSC component has an internal structure as depicted in figure 4. The GSC controller receives a call and executes the behavior as described in its configuration. Regarding the behavior, the GSC executes a certain scenario that consists of performing computational and / or memory consuming behavior. In addition, it can access a database executing a standard statement. Besides performing the behavior defined by the configuration, each GSC component has the ability to call other GSCs. In order to encapsulate the implementation of calling another GSC component, a set of caller with a unique interface is used. For each possible type of GSC component there exists one caller.

During all the activities the GSC performs it traces measurements. The measurements are stored asynchronously in order to keep the overhead as small as possible. After the simulation has finished the measurements can be collected via the adapter.

While in ABAP technology the GSC component is a function module, for the J2EE GSC component there exist two different types: the enterprise beans and the web components. The enterprise beans are implemented as described above. They can be called via the adapter or immediately by calling their remote interfaces. Web components are servlets and Java-Server-Pages (JSP). Usually a servlet receives the request and prepares the data for the JSP, which provides the response data. Thus, a servlet and a JSP appear as a pair and are represented as one component type in our framework.

The communication between a client GSC and the servlet-JSP-pair is done by requesting a URL on which the servlet is listening. The servlet then performs the action and returns the rendered JSP to the client GSC. By calling this URL out of a simple browser the performance engineer can experience the performance behavior, mainly the response time, as if he would interact with a real system.

B. Simulations With Implemented Components

Today it is not always the case that a project starts from scratch. Often an existing system needs to be modified or extended with some new components. The question regarding performance is then, how the new components will affect the whole system. Thus, it is obvious to include the existing parts in the simulation in order to get the best possible results.

In the pure GSC-based simulation, there was only one call configuration: a GSC calls another GSC. Including existing components in the simulation brings up two new scenarios: a GSC calls an existing component, and a real component calls a GSC.

In the first case, the challenge is to call the existing component in such a way, that it behaves like it would in the real system. This is not trivial, because the GSC has no application logic inside that produces these calls. It rather must choose a predefined call.

In the second case, when an existing component calls a GSC, the problem is more complex. The GSC component must be adapted for each call in order to implement the interface the real component normally calls. But in order to continue
in its usual behavior the real component needs a return value corresponding to the specific call, which is the more difficult part. Thus, the GSC component needs to know something about the application logic as it otherwise would not be able to return the corresponding values. The question is how much of the application logic the GSC needs to know, if not all.

The next step after integrating existing components in the simulation is incremental performance prediction. The idea behind it is to stepwise replace the GSCs by their corresponding real components. Always when a component is implemented it will be included in the simulation in order to narrow the simulation result to the exact performance behavior of the whole system. Although the simulation results based on the GSCs provide an important input for the development of the system, the results will never match exactly the later behavior. Thus, it is important to include as many real components as possible in the simulation.

V. Case Study

During the realization of the simulation framework, two case studies were used to gain the information needed for validation of the GSCs. The first one was an application we got from SAP, that was developed using ABAP-technology. The second one was the Java Pet Store from the J2EE BluePrints Program of Sun Microsystems [17] ported to the NetWeaver platform [18]. With both applications we did a reengineering in order to get the according performance model. Then we added traces to the applications in order to measure their performance. Finally, these measurements were compared with the simulation results.

A. Performance Model

The ABAP application is a simple batch application that analyzes a set of documents for duplicates. Its static architecture is depicted in figure 5. The boxes represent each one agent, which is realized by one GSC component during the simulation. The lines between the boxes (channels) show possible communication between the associated agents. Communication is possible in both directions if two agents are connected via a channel. The cylinders represent storages on which the agents can operate. If two agents operate on two different storages, it is guaranteed by the framework that no locking situation can occur. Otherwise, with a certain probability the two agents can lock each other.

The landscape information is not visualized within the diagram, instead a properties viewer is used to add the different properties to the model. Regarding the agents this is their location, the address of the application server, and the technology type the agent should be realized in. A storage is realized using a data schema consisting of different entities and relationships. As landscape information the number of instances for each entity can be specified.

Figure 6 shows the dynamic architecture of the ABAP-application. The model is based on the UML 2.0 Sequence Charts [19]. The boxes on top of the diagram represent the same agents as in the static architecture. Currently agents can only do synchronous calls to other agents, but asynchronous calls are planned. For each activity the performance relevant behavior can be defined by choosing a scenario out of a predefined set. We identified work on storages, algorithmic computation, and main memory consuming processes as the core parts of performance relevant behavior. The predefined scenarios consist of a combination of these parts.

The iteration of a set of calls to other agents can be managed using loops. In order to keep this model type simple, we decided not to support branches. On the abstraction level of architecture definitions, the assumption can be made, that only few branches are needed. Thus, the complete dynamic architecture can be described using some few call sequences with redundant parts.

The restrictions we made lead to simple process models that are easy to execute and allow for simple aggregation of the trace data. Our framework has implemented a lot of different aggregations that can be visualized within the diagrams. Figure 7 shows an example, where the results of a simulation of
the ABAP-application is visualized in the static architecture diagram.

Into each box three different types of performance figures can be projected. A fourth figure can be visualized using colors. The example shows the time the agent spent between incoming request and outgoing response aggregated for all calls, called busy time. Since this is a batch application, execution time is more relevant than response time. In addition, the number of incoming and outgoing calls is depicted. As the storages are passive components, no measurements are taken and thus, nothing is visualized. Regarding the channels, it is planned to measure and visualize the amount of data that is passed through the channel during the different calls.

Visualization of results is possible in the sequence charts in a way similar to the block diagram by adding numbers and using colors.
**B. Analysis**

Table I compares our measurements of the ABAP application to the simulations we ran within our framework. It shows the values for the average operating time (avg. OPT) of the individual components.

The simulation results shown in table I and figure 7 are a factor 2.6 lower than the measurements we took from the ABAP application. While this seems to be a big deviation, one must consider that all components have the same relative difference\(^1\). Therefore, it is still possible to identify hot spots in the architecture using our framework. If not the overall response time is subject of concern but the distribution of resource usage, our framework can even at this early stage provide useful results. Also, even if the simulation runs faster than the real application, it is very helpful for the architect to have an estimate of the final performance, which is relatively close considering that none of the application’s components has to be existing. Bad response time in the simulation would mean even worse response times in the application. In order to get simulation results that are closer to the measured values, the chosen performance model needs to be adjusted. Especially the activities the GSCs execute need further attention.

---

\(^1\)the aforementioned factor was constant for all parts

---

**VI. CONCLUSION**

In this paper we have presented a methodology for simulation-based performance engineering. With our approach, it is possible to get a feeling of a software system’s performance even in a very early stage of its development: when the high-level software architecture is being conceived. The software architect therefore creates high-level architectural diagrams as well as sequence diagrams that depict the main call scenarios between the software components. This architecture is then simulated by deploying GSCs representing the modeled software components into a real, up and running middleware platform. The modeled call scenarios are then executed, while performance metrics are being recorded. The results enable the software architect to identify potential hot-spots prior to actually implementing the software components. He can then resolve these hot-spots by refining the architecture, altering the models and re-running the simulation to evaluate his refinements.

We validated the applicability of our methodology by developing a framework that consists of a model editor with included visualizer and configurable GSCs for JAVA and ABAP technology.

We tested our frameworks with a case study, a batch processing software in ABAP. Our results so far show that our framework - in its current implementation - can be used to obtain first estimations of a software system’s performance. All measured performance metrics of the simulations deviate from the real systems by the same constant factor. Thus, bottlenecks in the system’s architecture can be identified by the relative distribution of computation time. Further fine-tuning of our GSCs to imitate more accurately the performance of the existing system will therefore be necessary, while our GSC component concept in general has proven to be reasonable.

Apart from the accuracy of the simulations conducted within our framework, our methodology presents further advantages over analytical performance prediction: As far as industrial applicability is concerned, performance prediction should be as simple as possible. High-level system architectures and the main call scenarios (i.e., the use cases) are artifacts that are produced anyhow when conducting a software development project, as opposed to maintaining formulae used to represent...
a specific performance behavior. Also, the infrastructure that the software components are embedded into is captured by the simulation. Furthermore, the infrastructure does not have to be modeled as real infrastructure platforms are used. This is especially valuable as modern middleware consumes a large share of the total execution time of a software system. The rising number of integration projects underlines the importance of middleware. In contrast, complex infrastructure platforms can hardly be described using analytical models, due to their complexity. Analytical models also have to capture different scenarios regarding the availability of resources such as CPU, main memory or network bandwidth. Our simulations in contrast can simply be run on different hardware setups. Therefore, our approach can also be used to test different deployment setups.

The use of real hardware and infrastructure is not free of drawbacks: First, the infrastructure (e.g., servers running the middleware) has to be in place already before the implementation and is required to reflect the system landscape intended for the later deployment of the software under development. This can be problematic especially when our approach is to be applied at very large software development projects, due to the fact that hardware is an expensive asset. Second, our approach builds on simulations that ideally run exactly as long as the software being under development. While this is appropriate for end-user applications, the simulation of batch processing applications may be time-consuming.

Our future work will not only include further refinement of our GSCs, but also the conduction of more case studies to confirm and further demonstrate the applicability of our approach. Moreover, research on how the GSCs could be replaced by real components throughout a software project in an incremental fashion has to be undertaken. The possibility to do so would enable the software architect to gradually verify the predictions produced with our methodology during the software development process. Also, the framework could be extended to support other middleware platforms and to provide dummies for component technologies other than ABAP and J2EE.

ACKNOWLEDGMENT

The authors would like to thank SAP for their support and valuable feedback during the development phases of this project.

<table>
<thead>
<tr>
<th>Main</th>
<th>avg. OPT (simulated)</th>
<th>No. of simulations</th>
<th>avg. OPT (measured)</th>
<th>No. of measurements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Creator</td>
<td>0.0 s</td>
<td>1</td>
<td>0.0 s</td>
<td>1</td>
</tr>
<tr>
<td>Duplicate Case Builder</td>
<td>44.7 s</td>
<td>30</td>
<td>0.9 s</td>
<td>30</td>
</tr>
<tr>
<td>Processor</td>
<td>0.5 s</td>
<td>35</td>
<td>0.8 s</td>
<td>39</td>
</tr>
<tr>
<td>Search Engine</td>
<td>125.0 s</td>
<td>20575</td>
<td>33.5 s</td>
<td>20551</td>
</tr>
<tr>
<td>Subset Check</td>
<td>12.5 s</td>
<td>1704</td>
<td>5.1 s</td>
<td>1703</td>
</tr>
<tr>
<td>Duplicate Updater</td>
<td>0.0 s</td>
<td>720</td>
<td>5.5 s</td>
<td>712</td>
</tr>
<tr>
<td>Duplicate Check</td>
<td>4.3 s</td>
<td>10269</td>
<td>3.5 s</td>
<td>10260</td>
</tr>
</tbody>
</table>

TABLE I
SIMULATION RESULTS VS. MEASUREMENTS

REFERENCES