Applying Software Performance Engineering Method to Development of Interactive Software*

YANG Yun [1,2], CHENG Jia-xing [1]

[1](*Key Laboratory of Intelligent Computing & Signal Processing, Ministry of Education, Anhui University, Hefei* 230039, *China*);

[2](*School of Information Technology, Swinburne University of Technology, Melbourne* 3122, *Australia*)

E-mail: cjx@mars.ahu.edu.cn

http://www.ahu.edu.cn

Received January 15, 2002; accepted August 3, 2002

Abstract: Performance issues are vital to the success of many interactive software systems, including software development tools and Web-based tools. However, the issue of performance engineering is still not emphasized in the process of software development. The key aim of this paper is to illustrate an effective method to achieve satisfactory performance for interactive software after careful design. This paper presents how to apply the software performance engineering (SPE) method by focusing on performance estimation at the design stage, and its effect on determining implementation approaches, in the development of interactive software tools. In addition to the rigid quantitative estimation method originated from SPE, this paper argues that the performance can also be cost-effectively estimated either semi-quantitatively or non-quantitatively. With the experience results described in this paper, it is suggested that it may not be compulsory to achieve direct quantitative performance estimation from environment specifications for every software development as advocated by the SPE method. This paper demonstrates a combination of analytical and experimental approaches to assessing the performance at early stages in development of software tools. It is hoped that this can reflect the essence of good experimental computer science.

Key words: software performance engineering; software design methodology; performance estimation; performance evaluation; interactive software

Software performance engineering (SPE) is advocated by Smith[1]. Performance refers to the response time as seen by the user. Lack of response limits the amount of work processed, so it determines a system's effectiveness and the productivity of its users. Many users subconsciously base their perception of (computer) service more on system responsiveness than on functionality. This phenomenon becomes more critical in a response-sensitive system such as an interactive system. For example, responsiveness is an essential quality-of-service attribute of Web applications. If a Web application is slow in responding to the requests, users will go elsewhere[2]. Unfortunately, up to now, software engineering has traditionally focused on functional requirements and how to build software that has few bugs and can be easily maintained. Work on the inclusion of performance engineering throughout the life-cycle had made relatively little impact[3]. Therefore, in fact, meeting performance requirements is as vital as meeting

* This work is partly supported by the Ministry of Education of China under the Visiting Scholar Scheme in 2001.

YANG Yun was born in 1963. He is an associate professor at the School of Information and Technology, Swinburne University of Technology, Australia. His current research areas are Internet computing and e-commerce. **CHENG Jia-xing** was born in 1946. He is a professor and doctoral supervisor at the Department of Computer Science, Anhui University. His current research areas are database and information systems.

requirements of functionality, reliability, and maintainability.

This paper addresses the issue of applying the software performance engineering method to development of interactive software with certain extension. The work presented in this paper has two case studies. One case study is based on a problem described previously[4,5] where a loosely-coupled tool interfacing paradigm was developed. During exploration of the loosely-coupled tool interfacing paradigm between concurrently executing user- and system-related tools, we perceived two general mechanisms of eager and lazy transmission for information transfer between tools. The other case study is based on another problem described previously[6,7] where a Web-based real-time cooperative editing tool was developed. During exploration of execution of local and remote editing operations in an Internet-based environment, we perceived some general mechanisms of eager and lazy multicasting of local edit operations to the remote sites and eager and not-so-eager execution of remote edit operations received. Based on SPE, a design choice is resolved by performance evaluation of the alternatives, but our experience suggests that it may not be compulsory to achieve direct quantitative performance estimation from environment specifications for every software development as advocated by the SPE method.

In a broader sense, this paper demonstrates a combination of analytical and experimental approaches to assessing the performance of different strategies in software development, by which, we hope, can reflect the essence of good experimental computer science. It is also intended to represent a theoretical discussion supported by some hard results. In subsequent sections, the relevant background of software performance engineering is summarized. Then two case studies are detailed in order to address the problem first, followed by design-stage performance estimation and concrete performance evaluation. Before the conclusion is drawn, some general comments on where the method can be systematically applied are discussed.

1 Software Performance Engineering

Historically, systems without critical performance requirements have adopted the **fix-it-later** method. It advocates concentration on software correctness, defers performance considerations to the integration-testing phase, and (if performance problems are detected then) corrects problems with additional hardware, with system and software “tuning”, or both. The rationale of fix-it-later was to save development time, expense, and maintenance costs. The savings are not realized, however, if initial performance is unsatisfactory because the additional time and expense for correcting problems are far greater than the cost of built-in performance which is a common sense in software engineering. Therefore, applying SPE at the early stages of software development is important[8] which would help the determination of appropriate mechanisms for software development.

As described by Smith[1], software performance engineering, the alternative to fix-it-later, is a method for constructing software systems to meet performance objectives. In principle, the process begins early in the software lifecycle and uses *quantitative* methods to identify satisfactory designs and to eliminate those that are likely to have unacceptable performance from the widest range of options, before developers invest significant time in implementation.

Some fundamental benefits of SPE are

- *increased productivity*

  when developers’ time is not invested in an implementation that must later be discarded and when implementation and testing focus on critical parts of software;
- *improved quality and usefulness of the resulting software product*

  by selecting suitable design and implementation alternatives, thus avoiding widespread tuning modifications which may compromise the initial design.

SPE performance assessment, which is applicable throughout the lifecycle, involves performance model

construction and evaluation for *best*, *average*, and *worst-case* analysis. Whatever technique designers select for development, it must explicitly specify the desired *behaviour* of the software (what it must do) and a *structure* for the proposed system that will realize that behaviour (i.e. a decomposition of the system into components and the function for each component). With SPE, when an appropriate decomposition is completed, some typical execution model is used to derive a *quantitative* performance prediction by applying environment specifications such as CPU speed, I/O device speed, ratio of high-level to machine instructions, to decomposed components.

Although it is a common practice to have comprehensive quantitative performance prediction, in this paper, we argue that it may not be essential to every software development. The case studies in the following two sections will demonstrate our experience with effective performance engineering.

2 Case Study — Applying SPE in an Interactive Software Development Tool

2.1 Problem background

Previously[4,5], we have discussed the loosely-coupled tool interfacing paradigm for software tool integration. In this section, our application of SPE is the development of that interfacing paradigm through the design and implementation stages[9]. In this section, we summarize the background.

Tool integration can be achieved by many different interfacing paradigms. Generally speaking, tight integration is a desirable property of a tool set, but it generally results in a closed and inflexible tool set. If we seek open and flexible tool sets, it is often at low levels of integration[10,11]. We have investigated three different paradigms ranging from the two extremes of uncoupled file-based tool integration and tightly-coupled internal representation-sharing tool integration, to the compromise of the loosely-coupled message-passing paradigm[4,5,12]. From a performance viewpoint, our experience with the uncoupled and tightly-coupled paradigms has shown that the tightly-coupled paradigm can offer the excellent response time while the uncoupled paradigm shows an unsatisfactory response. The general merits of designing the loosely-coupled paradigm have been clearly described previously, but here, we focus on the performance aspect. Our aim is to develop a loosely-coupled paradigm without significant performance degradation compared to the tightly-coupled approach.

The front-end is a generic language-based editor using a concrete syntax tree (CST) internal representation while each back-end is a tool normally using an abstract syntax tree (AST) internal representation to provide a service such as semantic analysis. In the loosely-coupled paradigm the front-end and back-end are separate tools running in parallel as two concurrent processes. Information transfer involved is based on message passing using inter-process communication (IPC). In contrast, the front-end and back-end in the tightly-coupled paradigm are linked together as a single executable program and share the same internal representation with conceptual separation only of the CST and AST. Compared to the tightly-coupled paradigm, it is clear that extra delays involved in the loosely-coupled paradigm result directly from information transfer. In this application, we concentrate on the performance bottleneck of representation transmission from the front-end to the back-end.

Basically, as introduced in previous papers, there are two different mechanisms for representation transmission in the loosely-coupled paradigm:

- *lazy transmission*

  where all changed units are transmitted to the back-end just before the back-end service is invoked, and
- *eager transmission*

  where after each user edit operation the changed unit or units are transmitted to the back-end immediately.

Which mechanism is more user-responsive needs evaluation as early as possible so as to choose the appropriate mechanism for implementation. The next subsection assesses the performance at the design stage from the SPE

viewpoint.

2.2 Performance estimation

In this subsection, we assume that the available CPU resource is a single processor on which the front-end and back-end are clearly asynchronous processes. We consider front-end and back-end overheads as contributing to user-observable delays additional to tightly-coupled delays for summarizing possible delays experienced by the user. Initially, we borrow the O-notation commonly used in the time complexity analysis for all decomposed components.

The front-end contributions to representation transmission are as follows: (1) cost of determining an AST unit to be transmitted including mainly the extraction cost of a CST unit traversal which effectively is $O(size_{ASTunit})$ because the ratio of CST unit size to AST unit size is more or less fixed for any grammar; (2) cost of encoding an AST unit which is the cost of encoding an AST unit involving computation cost of $O(size_{ASTunit})$; and (3) cost of transmitting an AST unit which involves the transmission IPC cost of $O(size_{ASTunit})$ in general. In summary, we can sum all contributions to calculate the total cost of the front-end. Therefore, we use k_{FE} which is a constant, and s_{FE} which is a measure of the *overall size* of the front-end change involved (i.e. the AST unit size), to arrive at the front-end extra delay as $k_{FE} * s_{FE}$.

Similarly, the back-end contributions are as follows: (1) cost of interpreting the encoded AST representation received including mainly the cost of receiving the corresponding AST unit which is similar to front-end transmission with a corresponding reception IPC cost of $O(size_{ASTunit})$ and the cost of decoding the corresponding AST unit which again involves a computation cost of $O(size_{ASTunit})$; and (2) cost of building the AST needed including the cost of constructing new nodes of the AST unit which is also similar to the front-end involving a computation cost of $O(size_{ASTunit})$ and the cost of possibly disposing of old nodes leading to a computation cost of $O(size_{old\_ASTunit})$. In summary, we can sum all contributions to calculate the total cost of the back-end as well. Similar to the front-end contribution, we use k_{BE} which is a constant, and s_{BE} which is a composite measure of the *overall size* of the back-end change involved, to arrive at the back-end extra delay as $k_{BE} * s_{BE}$.

Eager transmission distributes transmission to the completion of each edit operation and so offers the potential for back-end AST update in parallel with further user edit activity. Eventually when back-end service is required, all necessary transmission and update may be complete, so that service can be carried out immediately. In contrast, the lazy transmission mechanism adopts sequential processing without the potential for parallelism. The extra delay to the back-end request is the sum of front-end and back-end delays.

Table 1 User experienced extra delays

Transmission mechanism		Initial loading	Edit operation	Back-End request
eager	modal	$k_{FE} * S_{FE}$	$k_{FE} * s_{FE}$	0
	modeless			0 or $k_{FE} * s_{FE} + k_{BE} * s_{BE}$
Lazy		0	0	$k_{FE} * \Sigma s_{FE} + k_{BE} * \Sigma s_{BE}$

Table 1 summarizes the estimated extra delays experienced by the user via the eager (both modal and modeless, as explained later) and lazy transmission mechanisms based on typical cases — initial document loading, a typical insert operation, and a back-end request which are further explained later in this subsection. Similar arguments apply to other circumstances. In the table, S stands for the *overall size* of the **entire** AST which relates to initial document loading; s stands for the *overall size* of **one** AST update, which relates to the current edit operation typically involving a small amount of the document such as less than 50 lines of code – the normal size of a procedure in most programming languages; and Σs stands for the *overall size* of **one or many** AST updates since the previous evaluation. The relationship among the sizes is $s \le \Sigma s \le S$.

According to the discipline of SPE, the next step is estimation at the *quantitative* level to derive some magnitudes (in an absolute performance manner) using some specific environment specifications. Based on the above decomposition, it is feasible to collect environment specifications and estimate front-end and back-end contributions. However, in our

case, it seems more cost effective to proceed by indirect estimation for loosely-coupled integration from known performance figures from the tightly-coupled case.

Specifically, in the tightly-coupled paradigm, for example, as measured for a typical 500 lines of Pascal code on Sun 4/75, the responses to initial loading and corresponding back-end semantic analysis request are around 4 and 0.5 seconds respectively, and the response to each subsequent back-end request is proportional to user changes. Based on the magnitudes of the tightly-coupled paradigm, we analyze performance of the loosely-coupled paradigm. In detail,

- *initial loading*
  - A fully eager transmission strategy must transmit the entire document to the back-end tool as soon as it is loaded via the I/O channel involving extra delay of $k_{FE}*S_{FE}$. Assuming that no buffering delays occur, the time to transmit the entire document via IPC when it is first read in should be significantly less than the time of 4 seconds needed to parse the input and build the CST, since no parse costs are involved and the IPC is presumably a faster transmission mechanism than the source input, say via hard disk. Even if programs are stored in some efficient form, the transmit time should be less than the load time for the latter reason. In summary, although it may well be about 1 second in magnitude, the extra delay should be an acceptable fraction of the total.
  - No extra delay is involved in the lazy mechanism.
- *a typical insertion*
  - When an insertion operation is completed, the extra delay of $k_{FE}*s_{FE}$ will be a fraction of the initial load delay corresponding to the fraction of the overall program that is re-transmitted. If the re-transmitted unit is the block enclosing the change, and a 500 line of Pascal program is typically composed of at least 10 blocks, each 50 lines or less, then the corresponding extra delay is likely to be about 0.1 seconds. Delays of this magnitude are at the limit of user perceptibility[13], but in practice whether any such delay is noticed by the user may depend on whether or not the editor is modeless. In a modal editor, the user terminates insertion by an explicit key stroke or mouse click and then goes to the next operation. In this circumstance, there exists a precious time gap which the front-end can use to transmit the changed unit(s) to the back-end. In contrast, a modeless editor detects that the user is finished the current insertion only by the user's attempt to initiate the next operation. In this circumstance, eager transmission must contend with the user's need for immediate editor-response, and is more likely to cause a noticeable delay. In general, the amount of transmission information involved is small since it is only related to the current editing region. Therefore, whether the extra delay is noticed at all depends on the user and system activity at the time.
  - No extra delay is involved in the lazy mechanism.
- *back-end request*
  - When the back-end request is immediately after a modeless edit operation, an extra delay of $k_{FE} * s_{FE} + k_{BE} * s_{BE}$ is involved. The sum of the front-end and back-end delays is likely to be approximately twice the front-end delay since similar transmission and traverse/build patterns occur at either end. If it occurs, therefore, this delay is likely to be about 0.2 seconds as reasoned above. Depending on the subsequent analysis time involved, this may constitute a small but noticeable delay. For requests in other cases, no extra delay is noticed.
  - A fully lazy transmission strategy defers all representation transmission using (the relatively slow) IPC to the back-end request stage. While the user expects response of magnitudes of 0.5 seconds for the first request and better for subsequent requests which are available in the tightly-coupled paradigm, the extra delays of $k_{FE} * \Sigma s_{FE} + k_{BE} * \Sigma s_{BE}$ are effectively related to the size of the entire document for the first request and the size of all user changes without overlapping since the previous request. As reasoned above,

the total extra delay for the first request is likely to be about 2 seconds in magnitude which is large compared to the tightly-coupled analysis time of 0.5 seconds, and for each subsequent request the extra delay is likely to be about number-of-re-transmitted-blocks*0.2 seconds. In general, the number of blocks transmitted can be less than the number re-analysed, but, the extra delay is again likely to be large compared to the analysis time involved and discernible by the user.

In summary, because of the potential for parallel execution of the front-end and back-end processes, the eager approach can improve the user-observable delay for each back-end request without significant degradation of the editor performance at other times, particularly with modal editing. This case study, using semi-quantitative estimation, therefore shows the relevance of SPE in design-stage selection of the eager transmission implementation strategy, and in addition, our experience of using SPE also demonstrates that although appropriate decomposition is necessary, direct quantitative performance estimation from environment specifications may not be essential in all cases.

2.3 Performance evaluation

In this subsection we give an overall quantitative comparison of the tightly-coupled and loosely-coupled paradigms as determined experimentally from the prototype. The experiment measured CPU time performance. The front-end used is the Pascal modal editor and the back-end tightly-coupled and loosely-coupled Pascal incremental semantic analysers[14] are the same.

Figure 1 depicts our first experiment showing the CPU time consumed in direct response to specific user requests, using the two integration paradigms. The results for the loosely-coupled paradigm are based on the eager transmission strategy with one of four coordinate representation options described there[4] which, from the performance viewpoint, are similar because of only relatively small differences in performance among them. The results are shown for six different Pascal programs, characterised by the number of lines of code (LOC) in each. All times are shown in seconds, measured to an accuracy of ten milliseconds on a Sun 4/75. Each specific time interval is measured by recording and subtracting two CPU running times immediately before and after the task.

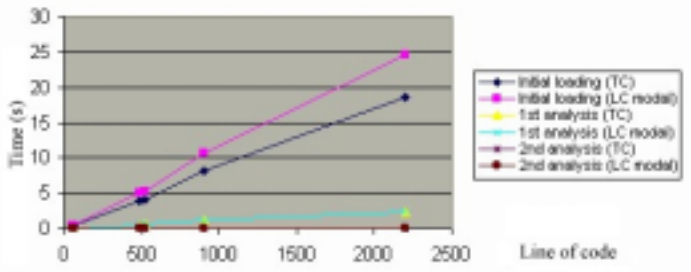

Fig.1 Performance curves

The *initial loading* time is the CPU time consumed by the front-end editor in first loading the program from a text file. For both paradigms this involves parsing the file concerned, but for the loosely-coupled paradigm it also involves eager transmission of an AST representation to the back-end analyser. (In both cases, the actual delay observed by the user may be greater due to hard disk activity during loading.) The 1st analysis time is the CPU time consumed (by the front- and back-ends) in analysing the whole program. The 2nd analysis time is the CPU time consumed for re-analysing the program without any intervening change.

The results in Fig.1 for the modal eager loosely-coupled paradigm show roughly a noticeable one third extra delay at initial load, but otherwise identical performance to that of the tightly-coupled paradigm. Given this consistency, we then focused on a typical case for further comparison of the two integration paradigms, namely, tightly-coupled and modal loosely-coupled, during editing operations. Table 2 represents a comparison for two integration paradigms which

indicates the CPU time consumed by five specific user requests on the 520 LOC Pascal program based on the same mapping facility. In addition to requests defined in Fig. 1, we have two extra requests. The *FE* 5 *changes* time is the CPU time consumed by the front-end for five user updates at five different points. The basic time X_i depends on the edit command and is irrelevant to our consideration. We note, however, that this time is increased by the time required for eager transmission of the resultant change in the loosely-coupled case. The 3rd *analysis* time is the CPU time consumed by the front- and back-ends in re-analysing the changed program immediately after the fifth change. In each case the block enclosing the change made is around 50 LOC (the typical size of a procedure). Given the block-based incremental processing strategy of the analyser concerned[14], the five changes together imply re-analysing about half of the program.

Table 2 Typical performance comparison and extrapolated performance

Interfacing paradigm	Initial loading	Loading ratio	1st analysis	Analysis ratio	2nd analysis	FE 5 changes	3rd analysis	Analysis ratio
TC	4.07		0.55		0.02	5*(*Xi*)	0.25	
LC modal	5.30	1.3	0.55	1	0.02	5*(*Xi*+0.08)	0.25	1
LC modeless	5.30	1.3	0.55	1	0.02	4*(*Xi*+0.08)+*Xi*	0.38	2.5
Lazy	4.07	1	2.55	4.6	0.02	5*(*Xi*)	0.90	3.6

The results show that with the loosely-coupled paradigm an extra delay of less than 0.1 seconds occurs after each edit operation due to eager transmission of the change involved. Such a delay is barely perceptible to a user in any circumstance. Typically, a pause of at least this magnitude occurs between user operations, so in practice the eager transmission delay is likely to remain completely undetected. Otherwise, the eager loosely-coupled paradigm achieves analytic response identical to the tightly-coupled paradigm.

At the moment, the modeless eager and lazy loosely-coupled paradigms have not been implemented, but existing data of the front-end and back-end extra delays can be used to extrapolate their performance. In Table 2, the times shown for the eager modeless and lazy approaches are extrapolated by adding the time consumed by back-end contributions, measured using the modal eager mechanism, to the front-end transmission and back-end analysis times shown for the modal eager mechanism. For example, in the case of the 3rd *analysis* for the eager modeless paradigm, the figure 0.38 is derived from a transmission time of 0.08+0.05, where 0.08 and 0.05 refer to front-end and back-end delays measured[4], and a back-end analysis time of 0.25.

Compared to the modal eager mechanism, the extrapolated results of the modeless eager mechanism differ in one minor way – the situation with initial loading is exactly the same, but the situation with editing is that the extra delay arising from eager transmission is now associated with the initiation of the next operation. Thus, the front-end extra delay for the fifth change is not included in the cumulative edit delays, but the corresponding front-end and back-end delays are added to the perceived time for the 3rd *analysis* request. In magnitude, however, the extra delay concerned (0.13 seconds) may or may not be noticeable depending on the user activities at the time.

In contrast, the extra delays imposed by the lazy transmission strategy on both the first and third analyses are clearly noticeable, particularly for the first request, where the response time is nearly 5 times that available with the tightly-coupled paradigm.

In summary, our experiments of prototypical implementation are in accord with the design-stage performance estimation carried out earlier.

3 Case Study — Applying SPE in a Real-time Cooperative Editor

3.1 Problem background

In a previous paper[6], we addressed our prototype of an Internet-based real-time cooperative text editor —— REDUCE (REal-time, Distributed, Unconstrained Collaborative Editing). REDUCE has a consistency model[7] that has

three properties: (1) *convergence* property, which ensures the consistency of the final results *at the end* of a cooperative editing session; (2) *causality-preservation* property, which ensures that dependent operations are executed in their natural causal-effect order *during* a cooperative editing session; and (3) *intention-preservation* property, which ensures that the effect of executing an operation at remote sites is the same as executing it at the local site at the time of its generation, and which also ensures that the execution effects of independent operations do not interfere with each other.

The consistency model imposes an execution order constraint only on dependent operations, leaving independent operations open as long as the convergence and intention-preservation properties are maintained. This feature lays the theoretical foundation for achieving good responsiveness by permitting local operations to execute immediately after their generation. Moreover, the intention-preservation property lets users see the effects of individual operations immediately while protecting against interference from other operations.

A REDUCE system consists of multiple cooperating sites. Each REDUCE site is typically a PC or a workstation connected to the Internet, with user interface facilities for displaying shared documents and generating editing operations; local storage facilities for storing replicates of shared documents; and computing and communication facilities for executing, synchronizing, and propagating editing operations.

In the context of Internet-based real-time cooperative text editing, we consider the primitive edit operations —— insertion and deletion of characters. At the design stage, however, we need to decide which implementation mechanisms should be used for the following cases:

- *eager or lazy processing for local edit operations*

  Eager means processing local edit operations at the character level of each key hitting for insertion and deletion, both reflected on display locally and multicast to the remote sites. In contrast, lazy means processing at the string level.
- *eager or lazy* (*not-so-eager*) *processing for remote edit operations*

  Eager means processing remote edit operations immediately regardless whether there are local edit operations in the queue. In contrast, lazy (not-so-eager) means local edit operations have higher priority.

Which mechanisms are more user-responsive in which circumstances need evaluation as early as possible so as to choose the appropriate mechanisms for implementation. The next subsection assesses the performance at the design stage.

3.2 Performance estimation

Rather than estimate quantitatively, we can analyse during the design stage to determine the implementation mechanisms. In the context of text editing, good responsiveness means that the user would not notice the delay, on the screen, of local edit operations of each character inserted or deleted. It is essential to have an eager processing strategy for local edit operations at the character level which is called *character-wise operation*. This will reduce the delay to the minimum.

Propagating each character-wise operation to remote sites over the Internet is not communication-efficient. In practice, cooperating users often work on different machines connected over the Internet with non-negligible and non-deterministic latency. While fibre-optic communication technologies can offer virtually unlimited Internet bandwidth, the communication latency over an intercontinental connection cannot be reduced much below 0.1 seconds due to the speed limit of light/electronic signals. It is therefore communication latency, rather than bandwidth, that challenges the design of Internet-based, response-sensitive systems and calls for latency-tolerant or latency-hiding technical solutions. So it would be better to have a lazy multicast strategy for accumulated local edit operations. Given this strategy, it is necessary to have an automatic character-to-string conversion scheme. When a character-wise operation is generated, it is saved in an internal buffer at the local site. The accumulated

character-wise operations will be converted into a so-called *string-wise operation* under some circumstances[6], for example, consecutively positioned characters without buffer overflow and time out. It is common sense in the distributed Internet environment that multicasting a string of characters is much more efficient than multicasting individual characters due to the communication overheads. This will also reduce the processing time at the remote sites simply because processing a single operation for a string would cost much less time than processing a number of operations for each character of the string.

In a cooperative editing session, there could be both local and remote edit operations in the processing queue from time to time. On one hand, the user expects to see the local edit operations immediately. On the other hand, normally, the users may not know what is happening at other sites and hence would not notice the delay of remote edit operations. To reduce the delay of local edit operations to the minimum, it is essential that the local operations have a higher priority than remote operations. Hence whenever a local operation has been generated, the longest possible waiting time for a local edit operation is bound to the time for completing a single remote operation, regardless of how many remote operations are in the queue. Otherwise, if an eager processing strategy is adopted for remote edit operations, a local edit operation may have to wait for all remote edit operations in the queue to be processed before it is processed which is most likely to cause noticeable delay depending on the number of remote edit operations for some occasions. Therefore, it is better to use a lazy or not-so-eager processing strategy for processing remote edit operations at a lower priority to the local edit operations.

The REDUCE consistency model specifies what assurance a cooperative editing system promises to its users and what properties the underlying concurrency control mechanisms must support. For causality-preservation, we used the well-known technique of state-vector–based time-stamping to selectively delay some dependent operations if they arrive out of causal ordering[15,7]. To support both convergence and intention-preservation under the constraints of high responsiveness, we devised an optimistic concurrency control technique, called operational transformation[15~17]. The novelty of operational transformation is that it allows independent operations to be executed in any order (hence local operations can be executed immediately) but ensures that their final effects are identical and intention-preserved.

In summary, although it is possible to derive quantitative performance estimation as advocated by SPE, in this case, we demonstrated how to apply the SPE principles to determine the implementation mechanisms at the design stage based on descriptive analysis and existing experience as non-quantitative estimation. Based on the analysis, it is sufficient to determine the implementation strategies at this design stage which is very cost effective. Next, we need to evaluate whether the strategies proposed are effective or not.

3.3 Performance evaluation

As indicated earlier, research has shown that users do not notice delays when response times are less than 0.1 seconds in interactive user interfaces. We conducted performance tests for the REDUCE system, which showed its response times to be well below this threshold of noticeable delay by implementing the mechanisms determined at the design stage based on performance estimation.

Our performance tests were conducted on a Sun Sparc 2/100 workstation. The time measurement is again accurate to milliseconds. As before, each specific time interval is measured by recording and subtracting two CPU times immediately before and after the related task or task sequence. For example, the CPU time consumption for processing a local insert is the time interval starting with the key event and ending when the character displays on the screen.

As shown in Fig.2, on average, the performance curves for local single-character insert and delete operations are far below the user noticeable delay mark of 0.1 seconds. This is true even if characters are typed at a very fast

pace. Note that the local deletion time is slightly longer than the local insertion time. This is because a REDUCE delete operation must collect and save the deleted text for future operational transformation against remote operations, while an insert operation has already known the inserted character when the key is pressed.

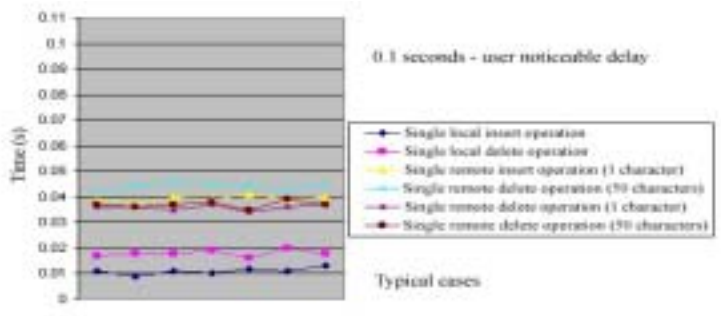

Fig.2 Performance of processing local and remote edit operations without interference

We also measured the execution time of remote string-wise operations to evaluate the benefits of propagating and executing them. In Fig.2, the performance for remote operations processing (including the time of operational transformation and execution) is depicted for string lengths of 1 character and 50 characters. Figure 2 shows that the time for processing a remote operation (either a 1-character or a 50-character string) is very short (less than 50 ms) and that the difference between processing a 1-character string and a 50-character string is very small. If the propagation strategy for the string-wise operation was not used, a 50-character string would cause 50 multicasting messages over the Internet and 50 1-character operation transformations and executions, which would be nearly 50 times slower than processing a single 50-character string-wise operation. In other words, the string-wise strategy is very effective in reducing the number of communication messages, transformations, and executions.

Based on the measurement of processing times for both local and remote operations, it is straightforward to evaluate the effectiveness of the scheme for setting a higher priority to the local operation-handling thread in REDUCE. Without such a priority strategy, the user would easily notice the delay of a local operation whenever remote operations were waiting in the queue, as illustrated in Fig.3. Under these circumstances, the response time for a local operation would be proportional to the number of remote operations waiting in the queue, and would cross the user-noticeable delay line when up to two remote operations were waiting.

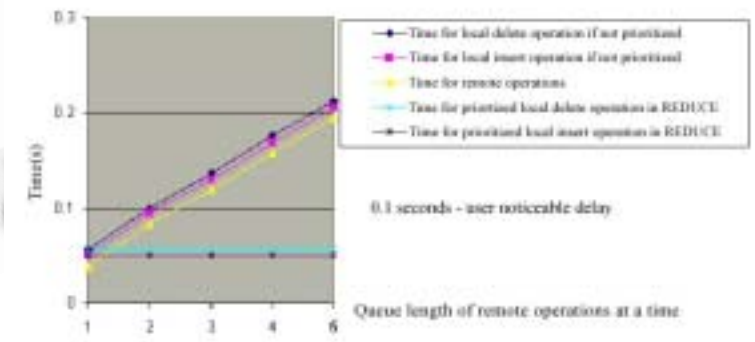

Fig.3 Performance comparison of local edit operation processing with interference

With a higher priority scheme in REDUCE, however, a local edit operation waits, in the worst case, only for the processing of one on-going remote operation, regardless of how many remote operations are waiting in the queue. This strategy results in a flat worst-case performance curve as shown in Fig.3, which is well below the user-noticeable delay line. In other words, it is essential to grant a higher priority for local operations, particularly

when a large number of remote operations have accumulated in the queue.

In reality, the response time of a local operation depends on the current load of the workstation as well as the CPU time consumed inside the REDUCE system. However, end-user computers are normally operated by single users with relatively light loads. When we measured the CPU time, we also measured the real-world time for editing operations. It is clear that only a fraction of extra time is added to the CPU time as the real response time. Therefore, the conclusions drawn from the performance results shown in Figs. 2 and 3 remain valid in real-world situations.

In summary, again, our experiments of prototypical implementation conform to the design-stage performance estimation carried out earlier.

4 Discussions

Based on the two case studies described in the paper, it is apparent that the principle of the SPE method can be applied flexibly in the following three ways:

- *Full-Quantitative performance estimation*
  This is the original method advocated for SPE which is used to derive a rigid quantitative performance estimation by applying environment specifications. With this method, a fairly accurate performance prediction can be made which is suitable and sometime essential for software with critical performance requirements. However, it is clear that this method is relatively complicated to use and hence costs the most among the three.
- *Semi-Quantitative performance estimation*
  This is the method which is most likely used to derive a relatively accurate quantitative performance estimation, particularly with an existing system which has some performance measurement available already. This method is relatively easy to use for performance estimation.
- *Non-Quantitative performance estimation*
  This is the method which can be used for performance estimation without quantitative figures. This method can be applied fairly easily to any software design in order to improve the performance but not necessarily quantitatively. It is clear that this method does not cost much at all.

In summary, all three kinds of estimation can be used at the software design stage for consideration of performance. Which one should be selected depends on the requirements of performance and the budget allocated.

5 Conclusions

In this paper, how to flexibly apply the software performance engineering method have been described, by which hopefully the essence of good experimental computer science has been reflected. The method enforces design-stage (quantitative) performance estimation for resolving design choices which has significance for development of response-sensitive software. The application of a loosely-coupled tool interfacing paradigm development has shown the effectiveness of this method. Based on semi-quantitative performance estimation at the design stage the eager transmission paradigm was determined to be a better and sufficient solution, which is in accord with prototypical implementation experiments. The other application of Web-based real-time cooperative editing on the Internet has also shown the effectiveness of simply apply the principle of the method. Based on non-quantitative performance estimation at the design stage, the eager processing for local edit operations, lazy processing for multicasting local edit operations and lazy (not-so-eager) processing of remote edit operations were determined to be a better solution, which is again in accord with prototypical implementation experiments. As an important extension to software performance engineering, this paper has demonstrated that although appropriate decomposition is necessary, direct quantitative estimation from environment specifications may not be essential to every software development. Therefore, direct quantitative, semi-quantitative and non-quantitative performance estimation can all be applied.

Acknowledgement We are grateful for all colleagues involved in the background publications for the two case studies used in this paper.

References:

[1] Smith, C.U. Performance Engineering of Software Systems. Addison-Wesley, 1990
[2] Pooly, R. Software engineering and performance: a road-map. In: Finkelstein, A., ed. The Future of Software Engineering. ACM Press, 2000. 189~199.
[3] Smith, C.U., Williams, L.G. Building responsive and scalable Web applications. In: Proceedings of the Computer Management Group Conference. 2000. http://www.perfeng.com/papers/webperf.pdf.
[4] Yang, Y. Tool interfaces for software development [Ph.D. Thesis]. Department of Computer Science, University of Queensland, Australia, 1992.
[5] Welsh, J., Yang, Y. Integration of semantic tools into document editors. Software, Concepts and Tools, 1994,15(2):68~81.
[6] Yang, Y., Sun, C., Zhang, Y., Jia, X. Real-Time cooperative editing on the Internet. IEEE Internet Computing, 2000,4(3):18~25.
[7] Sun, C., Jia, X., Zhang, Y., Yang, Y., Chen, D. Achieving convergence, causality-preservation, and intention-preservation in real-time cooperative editing systems. ACM Transactions on Computer-Human Interaction, 1998,5(1):63~108.
[8] Cheng, A.M.K., Clements, P., Woodside, M. Guest editors' introduction: workshop on software and performance. IEEE Transactions on Software Engineering, 2000,26(11):1025~1026.
[9] Yang, Y., Welsh, J. Software performance engineering — a case study for interactive software. In: Proceedings of the 16th Australian Computer Science Conference. Brisbane, Australia, 1993. 471~478.
[10] Lacroix, M., Vanhoedenaghe, M. Tool integration in an open environment. In: Ghezzi, C., McDermid, J.A., eds. Proceedings of the 2nd European Software Engineering Conference. Conventry, UK: Springer-Verlag, 1989. 311~323. Also in Lecture Notes in Computer Science, Vol. 387.
[11] Snodgrass, R., Shannon, K. Supporting flexible and efficient tool integration. In: Conradi, R., Didriksen, T.M., Warvik, D.H., eds. Proceedings of the International Workshop on Advanced Programming Environments. Trondheim, Norway: Springer-Verlag, 1986. 289~313. Also in Lecture Notes in Computer Science, Vol. 244.
[12] Yang, Y., Welsh, J., Allison, W. Supporting multiple tool integration paradigms within a single environment. In: Proceedings of the 6th International Workshop on CASE. Singapore: IEEE Computer Society Press, 1993. 364~374.
[13] Broom, B. Aspects of interactive program display [Ph.D. Thesis]. Department of Computer Science, University of Queensland, Australia, 1987.
[14] Kiong, D., Welsh, J. Incremental semantic evaluation in language-based editors. Software Practice and Experience, 1992, 22:111~135.
[15] Ellis, C.A., Cibbs, S.J. Concurrency control in groupware systems. In: Proceedings of the ACM SIGMOD Conference on Management of Data. ACM Press, 1989. 399~407.
[16] Ressel, M., Nitsche-Ruhland, D., Gunzenhauser, R. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In: Proceedings of the ACM Conference on CSCW. ACM Press, 1996. 288~297.
[17] Sun, C., Ellis, C.A. Operational transformation in group editors: issues, algorithms, and achievements. In: Proceedings of the ACM Conference on CSCW. ACM Press, 1998. 59~68.

软件性能工程方法在交互式软件开发中的应用

杨 耘[1,2], 程家兴[1]

[1](安徽大学 计算智能与信号处理教育部重点实验室,安徽 合肥 230039);

[2](Swinburne 大学 信息技术学院,墨尔本 3122,澳大利亚)

摘要: 在诸多交互式软件系统中,包括软件开发工具及基于万维网的工具,性能问题是至关重要的.然而在软件开发过程中,性能问题并未得到足够的重视.主要目的是展示一种有效方法,使得经过细致设计后交互式软件有满意的性能.展示如何应用软件性能工程方法于交互式软件工具开发中,特别注重在设计阶段的性能估算及其在决定实现方案时之效果.除了源于软件性能工程的严格的量化估算方法,提出性能亦可半量化或非量化方法进行有效而经济的估算.基于结果的描述,最终建议为源于软件性能工程方法学中苛求采用系统环境参数进行细化定量性能估算未必是惟一手段.演示了在软件工具开发早期阶段中一种用以评估性能的基于分析和实验的有效综合途经.希望能反映出了良好的实验计算机科学之本质.

关键词: 软件性能工程;软件设计方法学;性能估算;性能评价;交互式软件

中图法分类号: TP311 文献标识码: A