Usage History-based Architectural Scheduling*

Dongsun Kim, Seokhwan Kim, and Sooyong Park
Department of Computer Science and Engineering, Sogang University
Shinsoo-dong, Mapo-Gu, Seoul, Korea
darkrsw@sogang.ac.kr, goatkhan@gmail.com, sypark@sogang.ac.kr

Abstract

Waiting a long time for software applications to load typically elicits an adverse response from the user. This negative response eventually leads to decreased user satisfaction. The waiting time can be reduced by executing the application in improved hardware computing devices and by optimizing the algorithms constituting the application; however, these solutions are costly. An alternative approach is to overlap the execution and waiting times. Although this approach does not reduce the actual waiting time, it can reduce the user's waiting time. This study proposes an approach to decrease the waiting time by scheduling architectural units. The study formulates the dynamic architectural scheduling problem and it provides an overlapping approach to the problem on the basis of the formulation. This approach anticipates subsequent tasks from previous usage history and launches the corresponding components of the anticipated tasks in the task architectures. Evaluation of this approach shows that it effectively schedules applications and reduces waiting time.

1. Introduction

In spite of enhancements in contemporary computing systems, increasing the functions of software applications still leads to longer waiting times. Protracted waiting times result in low user satisfaction. One possible approach to decrease the waiting time is to use faster, improved computing devices; however, this leads to increased cost. Another possible approach is to overlap the application's execution and preparation times (i.e., loading and initialization time). Although this approach cannot fundamentally reduce the loading and initialization time of software systems, it can reduce the user's waiting time.

Another possible reason for an increase in waiting time is memory swapping. In a system, when the memory space used by the applications in a system exceeds the physically provided memory size, the system uses secondary storage (e.g., magnetic disks) as the memory space. This causes repeated storage accesses, which may lead to delays in loading. Therefore, this study provides a scheduling method for an application in a limited memory space.

In this study, we formulate the abovementioned problem into a dynamic architectural scheduling problem. This formulation defines tasks, task architectures, and components. The objective of this formulation is to minimize the waiting time by optimally scheduling the software architecture. On the basis of the formulation, an overlapping approach to decrease the waiting time is proposed. The approach provides a prediction algorithm that anticipates the subsequent user tasks in a short time. Although the prediction results are not optimal, the approach provides sufficiently good results without increasing the user's waiting time.

After anticipating the subsequent tasks, this approach loads and initializes components in the task architectures. During this period, newly loaded and existing components can exceed the specified memory space. Our approach applies a replacement algorithm based on the least recently used (LRU) principle. This algorithm removes components that were used a long time ago and maintains the application without exceeding the specified memory space. On the basis of the prediction and replacement algorithms, this approach deals with the dynamic architectural scheduling problem.

The remainder of this paper is organized as follows: Section 2 relates this approach to existing work. Section 3 provides a motivating example that requires dynamic architectural scheduling. Section 4 formulates the dynamic architectural scheduling problem. In Section 5, we propose our approach to the dynamic architectural scheduling problem; that the proposed approach comprises prediction and replacement algorithms. Section 6 evaluates our approach in terms of accuracy and applicability. Finally, Section 8 provides the conclusion and suggests further studies.

*This research was performed for the Intelligent Robotics Development Program, one of the 21st Century Frontier R&D Programs funded by the Ministry of Knowledge Economy (MKE).

0730-3157/09 $25.00 © 2009 IEEE
DOI 10.1109/COMPSAC.2009.65

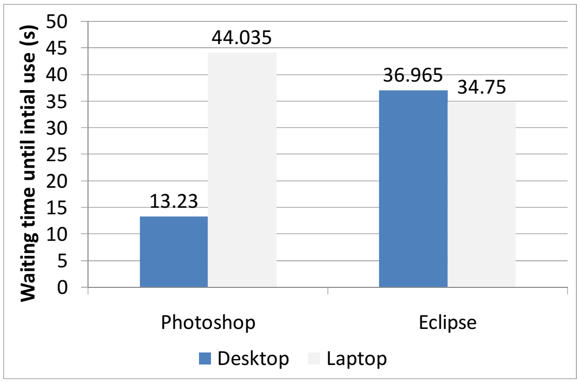

Figure 1. The waiting time until initial use of each application (Photoshop and Eclipse). Each time value is the average of ten trials. These values are measured on a desktop with 2.4 GHz CPU and 2 GB RAM and on a laptop with 900 MHz CPU and 1 GB RAM.

2. Related Work

Many researchers have investigated the literature on waiting time. On the Web, download speeds of web pages are primary design criterion [16]. Long download time leads to user dissatisfaction. Acceptable waiting time is sill controversial [1]. Nielsen [15] and Zona Research [26] advocate the 10-second limit and the 8-second rule, respectively, while other researchers present the two-second rule [23] and the 12-second rule [5]. Nielsen proposes the influences of various response times [13], for example, longer delays than 10 seconds may cause user dissatisfaction and users may need to be given feedback indicating that they should wait.

Some research on waiting times present the influence of users' intolerance in waiting for computer response. Miller [10] suggested the two-second rule based on human short-term memory. This suggestion advocated waiting time more than two seconds may lead to interference with human short-term memory. Nielsen [14] suggests that the waiting time should be less than one second for tasks in which uninterrupted focus is crucial while it should be less than 10 seconds for other types of tasks [13]. In Shneiderman's review [23] computing systems should give a response to users within two seconds.

Several research on user preference characterization suggest that applications should adapt its behavior and structure to the user's preference. Schiaffino and Amandi [21, 20] propose polite personal agents who characterize user profiles and provide personalized, context-aware assistance. Weld *et. al* [24] presents personalized user interfaces. This approach anticipates user behavior and provides user interface adaptation. Leung *et. al* [7] provides an approach to developing an adaptive user interface in Eclipse. This approach suggests a usage prediction mechanism that anticipates the sequence of usage and the Fade algorithm that determines which elements to be removed from the menu.

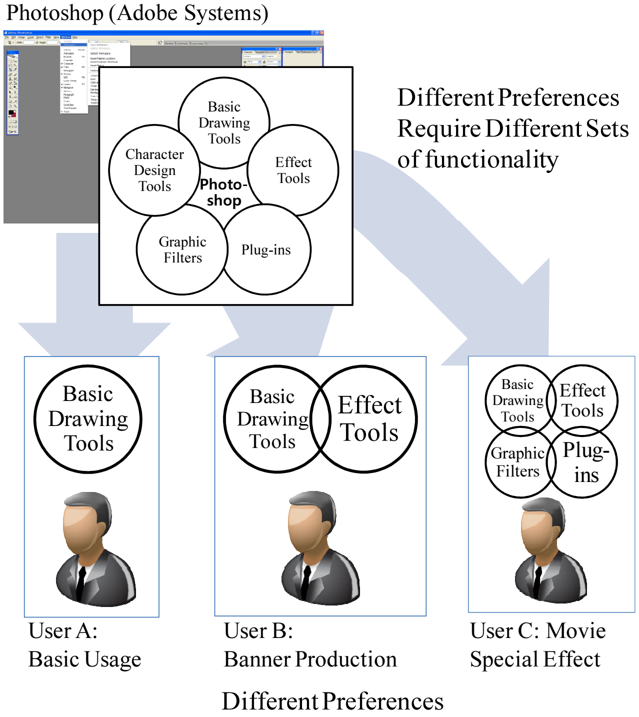

Figure 2. Different users have different preferences for applications.

3. Motivating Example

The amount of time users wait for computer applications may influence their satisfaction with those applications. For example, when a user is trying to edit pictures using an editing tool such as Photoshop or Paint Shop Pro, he or she must click the application's icon and must wait for the application to load. This waiting time is caused by preparation steps such as software module loading and the initialization of components. The loading time represents time needed to load the software module to the memory space. The initialization time represents duration of initializing the software module for use. A significantly long waiting time may elicit an adverse response from the user [20], and ultimately, it affects the user's satisfaction with the application.

According to [12], the maximum tolerable waiting time for the user is approximately two seconds, because of the problem of loss of human short-term memory after two seconds[10]. However, most popular applications have a

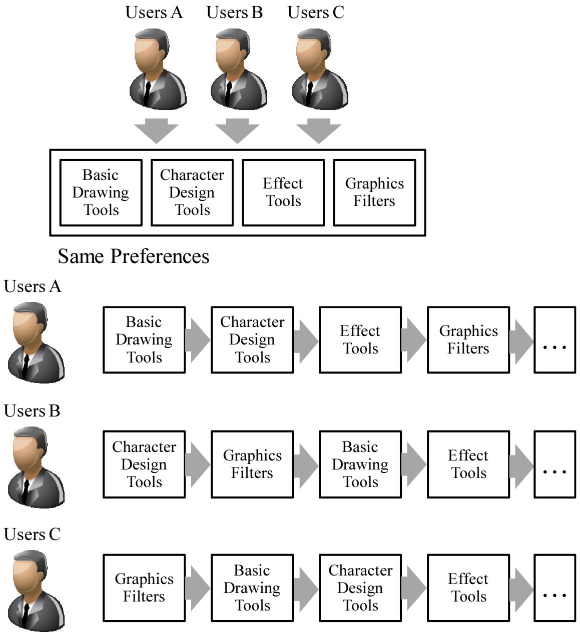

Figure 3. Different users perform their tasks in different sequences.

longer waiting time, as shown in Figure 1, which shows the time duration for startup of applications such as Photoshop and Eclipse to be definitely longer than two seconds. To reduce that time, on-demand loading of the dynamic linking library and dynamic shared objects can be used; however, those technologies merely divide the waiting time into smaller periods, and the total sum is the same. Moreover, on-demand loading results in too-frequent delays during execution. Therefore, it is necessary to reduce the entire waiting time.

Different users prefer different sets of functionality in an application. For example, User A uses only the basic drawing tools of Photoshop; User B mainly uses basic drawing tools and effect tools; and User C often uses basic drawing tools, effect tools, graphics filters, and other plug-ins, as shown in Figure 2. Similar patterns occur when users use different applications such as word processors and programming environments. This implies that all users' waiting times can be reduced if the application can infer the users' preferred sets of functions. In other words, the waiting time can be decreased if the application loads the required functions for a specific user.

Users achieve their objectives by sequentially executing their tasks. This implies that not all component tasks are required to be loaded because, in general, a user performs one task at a specific moment. For example, different users perform their tasks in different sequences, even if they have the same preferred set of tasks, as shown in Figure 3. If the application can anticipate the user's sequence of tasks, it can reduce the waiting time by immediately loading the required functions.

Loading every function of an application may require users to wait for prolonged durations; this is because excessive disk access due to page replacement leads to performance degradation if the application's memory usage exceeds the physical memory of the system. In other words, when the application uses an exceedingly large amount of memory space and some of them are located in the swap space, this may lead to repeated disk access and, in consequence, degradation in the application's performance. This implies that the application needs to minimize the memory usage in order to minimize the waiting time. Thus, functions unused for a long time should be removed from the memory space to minimize the usage of swap space.

The above-mentioned examples imply that the application must anticipate the user's task sequence, dynamically load the required functions, and optimize the memory usage to minimize the user's waiting time. The next section formulates this problem.

4. Dynamic Architectural Scheduling

This section formulates the dynamic architectural scheduling problem. The formulation defines tasks, task architectures, and components to solve the problem. The objective of this formulation is to minimize the user's waiting time and not to exceed the specified memory usage. This formulation is based on the following assumptions.

- Components are sufficiently divided in terms of memory usage.
- The waiting time of an application is fairly spread over its components.
- The application is implemented by dynamic architectures.

The first assumption implies that memory-usage efficiency can be influenced by one component if that component occupies most of the application's memory usage. Therefore, components that comprise the application should be decomposed so that a few components do not influence the efficiency. The second assumption is similar to the first assumption. The waiting time can be influenced by one component when that component consumes most of the application's total waiting time.

The last assumption implies that the application must be implemented by dynamic architectures such as Darwin [8], Acme-based architecture [4], C2-style dynamic structure [18], and Weave [17]. This enables the application to dynamically change its architecture when loading functions (components) into the memory space and removing them from it according to the schedule.

An application task is a unit of the user's action. The user performs a sequence of tasks to achieve his or her objectives. An application has a finite set of tasks that it can perform. This formulation assumes that a user performs a task for each time period. Task T_i has a software architecture [22] composed of a set of components to perform the task. Task architecture A_i represents a software architecture corresponding to task T_i.

Task architecture A_i organizes components and connectors that provide functionality to execute the corresponding task T_i. A task architecture has a subset of the entire architectural configuration of the application. Therefore, task architecture A_i can be formulated by

$$A_i \subseteq A_e$$

where A_e represents the entire architectural configuration of the application. This partial configuration should be able to independently deal with a sequence of actions specified by the corresponding task. In other words, the task architecture of a specific task must contain a sufficient set of components to perform the task.

A component in a task architecture is an element that contains executable code that actually performs the task. Components actually consume memory space and cause delays. Component C_i is formulated by three properties: memory usage, loading time, and initialization time. Memory usage of C_i is a function defined by time t as follows:

$$M : C \times T \rightarrow \mathbb{Z}^+$$

where M, C, T, and $\mathbb{Z}^+$ represent memory usage function, component set, time domain, and positive integer set, respectively. $M(C_i, t_k)$ returns the current memory usage of component C_i on time t_k in kilobytes. The loading time and initialization time of component C_i are denoted by $L_t(C_i)$ and $I_t(C_i)$, respectively, and defined by

$$L_t : C \rightarrow \mathbb{Z}^+$$

$$I_t : C \rightarrow \mathbb{Z}^+$$

where these functions return the loading time and initialization time of component C_i in milliseconds ($\mathbb{Z}^+$). These time measures are not time-dependent, because the functions return the average time measured in the development time. These measures can vary for each loading and initialization process; however, the variance is not large and the prediction of the loading and initialization times can be dependent on various properties. Therefore, the functions return the previously measured time values.

The objective of this problem formulation is to minimize the user's total waiting time by not exceeding a specified amount of memory space. The total waiting time indicates time elapsed during a sequence of task execution steps (i.e., the waiting time from the beginning to the end of the application). The total waiting time W_{tot} can be defined as

$$W_{tot} = \sum_{i}^{n} W_i$$

$$\begin{aligned} W_i &= \sum_{j}^{m} W_{C_j} \cdot E(C_j) \\ &= \sum_{j}^{m} [L_t(C_j) + I_t(C_j)] \cdot E(C_j) \qquad (1) \end{aligned}$$

where W_i is the waiting time to begin with *i*-th task, W_{C_j} is the waiting time of component C_j, and n is the number of tasks from beginning to end of the user's job. Every component C_j belongs to task architecture A_i of the currently requested task T_i, and m is the number of components that belong to A_i. $E(C_j)$ is a function that determines whether component C_j is already loaded in the memory space and defined as

$$E : C \rightarrow \{0, 1\}$$

where $E(C_j)$ returns 0 if component C_j is loaded, otherwise it returns a value of 1. The total memory consumption of currently loaded components (i.e., M_{tot}) can be specified as

$$M_{tot} = \sum_{k}^{l} M(C_k, t_i)$$

where k is the number of currently loaded components in the specified memory space. M_{tot} indicates the amount of memory space consumed by the currently loaded components when the *i*-th task is requested (i.e., time t_i).

Based on the above formulation, we can specify the objective of the dynamic architectural scheduling problem as

$$\text{minimize} \quad W_{tot} \qquad (2)$$

$$\text{subject to} \quad M_{tot} \leq M_{max} \qquad (3)$$

where M_{max} represents the size of the specified memory space for the application. Equation (2) represents the objective function, implying that the application must minimize the total waiting time to prevent user dissatisfaction, and Equation (3) represents the constraints, implying that the memory size consumed by the currently loaded component should exceed the specified memory space for the application. On the basis of the above objective and constraints, the next section describes our approach to dynamic architectural scheduling.

5. Approach

This section describes an approach to the dynamic architectural scheduling problem. The basic idea of this approach is to overlap task execution and component loading. The application cannot reduce the waiting time (i.e., loading and initialization times) of each component without fundamentally modifying the component implementation; however, such modification may impose additional development costs, or may not even be possible. Therefore, the remaining solution is to set $E(C_j)$, shown in Equation (1), to 0. In other words, the total waiting time can be reduced if the application can anticipate the sequence of the user's tasks and prefetch the required components prior to the user's task requests.

5.1. Task Prediction

To overlap task execution and component loading, our approach anticipates the user's subsequent tasks based upon his or her previous usage history. This approach accumulates the user's task-execution history and infers possible subsequent tasks based on the history. Since the objective of dynamic architectural scheduling is to minimize the waiting time, the inference must be performed quickly. Further, the prediction must be as precise as possible.

Task-sequence inference has properties similar to those of process scheduling for operating systems (OS). In process scheduling for OS, optimal scheduling algorithms such as shortest job first can be applied; however, this cannot be applied practically, because OS cannot precisely anticipate the execution time of processes. Moreover, the application cannot anticipate the sequence of the user's task. Therefore, the inference efficiency increases as the user performs similar sequences of task execution.

To infer the user's next tasks, the application must record and accumulate the user's task execution history from the beginning to the end. In the accumulation step, the application records every task in the execution. Then, it counts the number of immediately subsequent steps for every task. For example, when the application finishes the user's tasks, the application counts the number of occurrences of immediately subsequent tasks as shown in Figure 4. In other words, a table for the task inference is updated by counting the next steps of a task, based on previous sequences of task execution including the sequence executed right beforehand.

On the basis of the inference table, our approach anticipates the next tasks. For a given inference table and the current task requested by the user, this approach selects the most frequently occurring subsequent tasks as tasks to be prefetched. The number of tasks to be prefetched is determined by window size ω. For example, when the current task is task A and the window size is 3 ($\omega = 3$), the application selects tasks B, D, and H to be prefetched at a run-time based on the currently given inference table shown in Figure 5.

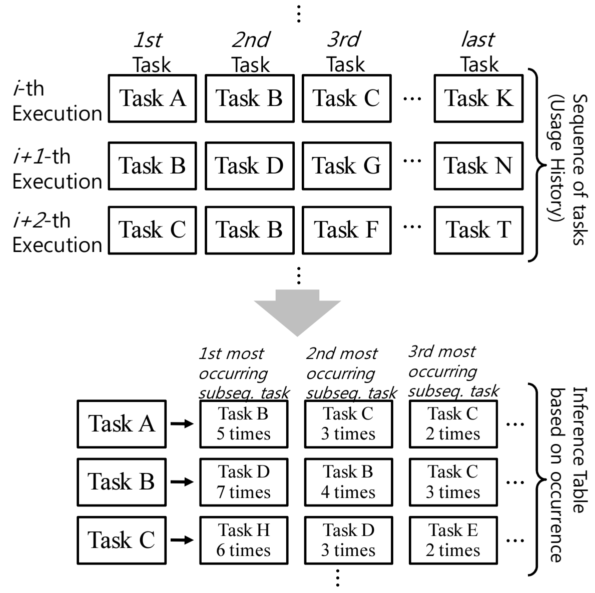

Figure 4. Task history accumulation and inference table building.

This prediction approach anticipates future tasks by table lookup. Although this approach cannot guarantee optimal scheduling, simple table lookup guarantees rapid scheduling. This rapid scheduling can be achieved by table building after the application ends. Table building is a time-consuming task, because it scans every previous task-execution history; however, this task is performed after application execution, and it does not affect the total waiting time.

5.2. Architecture Loading

The next step of this approach is architecture loading. On the basis of tasks anticipated by the previous prediction step, the application loads the corresponding task architectures to the tasks. This implies that it actually loads components in the corresponding task architectures. For example, suppose that the currently loaded component set C_{cur} is $\{C_2, C_4, C_5, C_6, C_7, C_{13}\}$ and the requested component set C_{req} is C_4, C_8, C_9, C_{12}. At this request, C_4 is already loaded and C_8, C_9, and C_{12} need to be newly loaded. Eventually, $\{C_2, C_4, C_5, C_6, C_7, C_8, C_12, C_{13}\}$ is the required component set. However, the dynamic architectural scheduling problem described in Section 4 specifies the maximum amount of memory space that the application can use. Therefore, it cannot load all required components

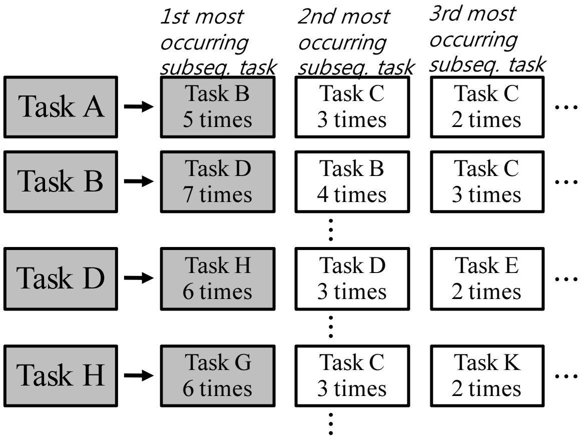

Figure 5. An example of task inference.

in the memory space.

To deal with this space constraint, our approach applies a replacement algorithm. Research in several fields such as page replacement, document replacement, and processor replacement has already yielded a number of replacement algorithms such as adaptive replacement cache (ARC) [9], least recently used (LRU), GreedyDual Size (GDS) [2], least frequently used (LFU), most recently used (MRU), and Pseudo-LRU (PLRU). Reineke *et al.* [19] compared a set of replacement algorithms from the perspective of behavior prediction, and their results implied that LRU is the best algorithm. Therefore, our approach adopts LRU as a component replacement policy.

According to LRU, the least recently used component must be removed from the specified memory space. To perform this, the application must maintain a table that records the last-used time of every component. When replacement is required, the application unloads components until the memory constraint is met. From the above example, C_4, C_8, C_9, and C_{12} are marked as MRU components because they are the set of newly requested components. Then, it sequentially removes the LRU components from component set $\{C_2, C_4, C_5, C_6, C_7, C_7, C_8, C_{12}, C_{13}\}$. Since the decision on LRU components can be made in a linear time, the algorithm does not lead to an increase in waiting time.

When the required components are not in the memory (i.e., a prediction is missed), eventually, the application must load the components on demand. This inevitably leads to user waiting time. This is the worst case, resulting in delays, because the user must wait for the preparation of software components when he or she simply needs to use them. This implies that the accuracy of the prediction algorithm directly influences the performance of the approach. Therefore, the approach must minimize the number of predictions missed.

Using the task-prediction and component-replacement mechanisms suggested by our approach, an application can minimize the user's waiting time by overlapping the execution and waiting times. The next section evaluates our approach from the viewpoints of prediction efficiency and applicability.

6. Evaluation

This section provides a k-fold cross-validation for the prediction algorithm and a case study to evaluate the applicability of our approach. The validation is conducted to verify how efficiently the prediction algorithm anticipates subsequent tasks. The case study shows the results of applying our approach to two open-source applications.

6.1. Prediction Accuracy

The waiting time can decrease as the efficiency with which the application anticipates subsequent tasks increases. This section verifies the accuracy of our task-prediction approach described in Section 5.1. In this section, prediction accuracy indicates how precisely the approach anticipates the user's next task. Prediction accuracy is measured as follows:

$$\text{Precision Accuracy} = \frac{\text{Prediction Hit}}{\text{Total Prediction}}$$

Our approach's prediction algorithm is evaluated by the k-fold cross-validation technique [11, 25]. This technique divides sample data into K subsets. Then, one subset is used as test set and $K - 1$ subsets are used as training sets. The technique repeats this test procedure for every subset. This evaluation employs a two-fold cross-validation (i.e., $K = 2$) because the size of the sample data is not sufficiently large.

The data for this evaluation is the user's task history, sampled from an undergraduate student's Photoshop activity. Independent functions such as plug-ins, drawing tools, and layer functions are defined as tasks; 22 tasks are defined and 50 sequences of tasks are recorded. The average length of sequences is 55. The window size varies from 1 to 22. The result is shown in Figure 6.

The lower (darker) and upper (lighter) areas represent the miss ratio and hit ratio, respectively. Hit ratio represents prediction accuracy, because hit ratio is calculated by `Prediction Hit` / `Total Prediction`. Hit ratio is larger than 80% from $\omega = 4$. This implies that the prediction approach effectively anticipates the subsequent tasks with small window sizes. Further, hit ratio converges to 100% as the window size increases; however, larger window sizes may lead to increased component replacement, because a large number of tasks may cause larger memory consumption. Therefore, it is important to determine

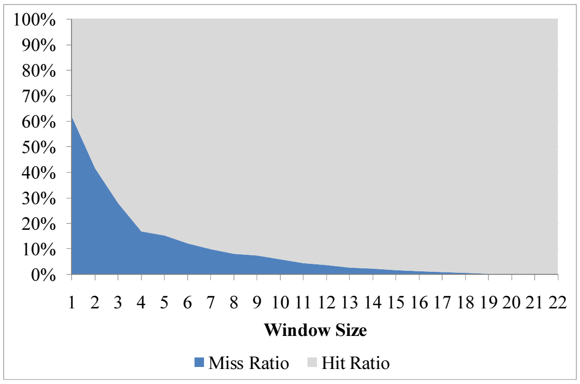

Figure 6. Miss ratio and hit ratio of the prediction approach described in Section 5.

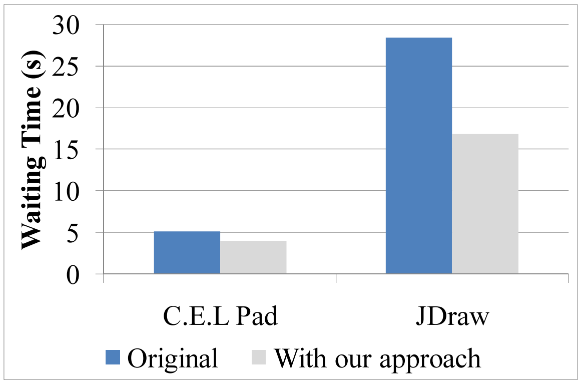

Figure 7. The comparison of original waiting time and the waiting time after applying our approach.

an appropriate window size. Further discussion about the window size is provided in Section 7.

6.2. Case Study

Two applications from the open-source community have been chosen to apply to our approach-C.E.L Pad [3] and JDraw [6]-because these applications have several options for user interaction. We re-engineered these applications to contain tasks, task architecture, and components. C.E.L Pad has 12 tasks and 26 components, while JDraw has 46 tasks and 103 components. This experiment is conducted on a desktop equipped with 800 MHz CPU and 512 MB memory.

The comparison between original waiting time and waiting time after applying our approach is shown in Figure 7. In this case study, the window size is four. C.E.L Pad and JDraw are provided 500 KB and 1 MB memory space, respectively. As shown in the figure, the waiting time of C.E.L Pad and JDraw is reduced by approximately 22% and 41%, respectively. This implies that our approach effectively reduces the waiting time.

7. Discussion

The first issue is the accuracy of the loading and initialization times. In this study, our formulation described in Section 4 assumes that the application is provided with information on the loading and initialization times of the components. However, it is difficult to provide precise values of loading and initialization times, because they can vary due to diverse factors such as the influence of installed devices (e.g., CPU, hard disks, and memory) and the current network throughput. In general, improved hardware devices provide enhanced performance; however, such devices are more expensive. Further, small devices such as cellular phones and PDAs have lower computing power; therefore, complex applications may lead to longer waiting times. Thus, it is necessary to apply an approach to reducing the waiting time by overlapping the execution and waiting (loading and initialization) times.

Our approach assumes two main issues: 1) the application can distinguish between different users and 2) each user executes the application by using similar sequences of tasks. The first assumption can be realized simply by user management of OS. Most OS provide user management functions and applications that can distinguish the current user. The second assumption is more complex. Several studies advocate that it is possible to identify the user's preference or profile of application usage. This implies that the application can anticipate a possible set of functions. However, this does not mean that it is possible to predict the future sequence of the user's tasks. Thus, it is necessary to conduct an in-depth study for usage sequences.

The window size is a crucial variable to precisely and efficiently anticipate the next subsequent task and use the specified memory space for the application. The experiment described in Section 6.1 shows that a larger window size improves the prediction accuracy; however, a larger window size decreases the memory-usage efficiency. Therefore, it is necessary to search for an optimal window size. An optimal window size for an application depends on the size of the given memory space, the number of tasks and components, and the consistency of usage history. Thus, it is difficult to search for the best window size for each application and each user. A possible approach to determine an appropriate window size is to investigate various users' usage history and adaptively specify the window size based on the consistency of the history. In other words, if the user history is highly consistent, then a small window size should be chosen; otherwise, a larger window size should be chosen.

In this study, the basic idea of our approach is to overlap the user's task-execution time and waiting time. This idea can be improved when the system provides a multi-core processor. If the application exploits a multi-core processor, it can physically overlap the execution and waiting times. Although a single core processor can perform the same overlap (because loading is more relevant to I/O time), execution and initialization times cannot be physically and fully overlapped in a single-core context. Fortunately, contemporary PCs (most desktops and a large number of laptops) have multi-core processors.

8. Conclusion and Future Work

Long waiting times elicit adverse reactions from users. Although the power of computing devices is increasing, software applications still take significant time to load and initialize their functions, because their requirements for computing power also are increasing. The user's negative response to long waiting times may decrease user satisfaction.

Decreasing the waiting time of an application is not simple. The waiting time comprises the loading and initialization times of components; therefore, it is difficult to optimize the time required to prepare the execution of components. Therefore, our approach overlaps the waiting and execution times to reduce the user's waiting time. This approach is implemented by anticipating the user's subsequent tasks. Then, it prefetches components of the predicted tasks.

The evaluation of our approach shows that an appropriate prediction can reduce the waiting time. It is important not to take excessive time to anticipate subsequent tasks because time-consuming prediction leads to increasing the waiting time. The prediction algorithm of our approach anticipates subsequent tasks from the user's task history in a short time, even though it has a high miss ratio with small window sizes. Further, case studies show that our approach can be used in practical applications.

Further improvements include the determination of an appropriate task, task architecture, and component size for an application. These sizes influence the performance of our approach. Excessively small sizes may lead to greater prediction complexity. Larger sizes may lead to the performance degradation of our approach. Therefore, it is important to divide tasks and components into appropriate sizes.

References

[1] B. Bailey. Acceptable computer response times, April 2001.

[2] P. Cao and S. Irani. Greedydual-size: A cost-aware www proxy caching algorithm. In *In 2nd Web Caching Workshop*, 1997.

[3] C.E.L. Pad. http://sourceforge.net/projects/cel-pad/.

[4] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[5] J. A. Hoxmeier and C. DiCesare. System response time and user satisfaction: an experimental study of browser-based applications. In *Proceedings of the Americas Conference on Information Systems, 10-13 August 2000 (Long Beach, California: Association for Information Systems)*, pages 140–145, 2000.

[6] JDraw. http://jdraw.sourceforge.net/index.php.

[7] A. Leung, S. Morisson, M. Wringe, and Y. Zou. Developing an adaptive user interface in eclipse. In *Proc. the Eclipse Technology eXchange Workshop at European Conference on Object Oriented Programming*, Nantes, France, July 2006.

[8] J. Magee and J. Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, New York, NY, USA, 1996. ACM.

[9] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *In Proceedings of the 2003 Conference on File and Storage Technologies (FAST)*, pages 115–130, 2003.

[10] R. B. Miller. Response time in man-computer conversational transactions. In *AFIPS '68 (Fall, part I): Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277, New York, NY, USA, 1968. ACM.

[11] A. W. Moore. Cross-validation, 2005.

[12] F. F.-H. Nah. A study on tolerable waiting time: how long are web users willing to wait. *Behaviour and Information Technology*, 23(3):153–163, 2004.

[13] J. Nielsen. *Usability Engineering*. Morgan Kaufmann, 1994.

[14] J. Nielsen. Guidelines for multimedia on the web. *World Wide Web J.*, 2(1):157–162, 1997.

[15] J. Nielsen. The need for speed, March 1997.

[16] J. Nielsen. User interface directions for the web. *Commun. ACM*, 42(1):65–72, 1999.

[17] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[18] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 177–186, Washington, DC, USA, 1998. IEEE Computer Society.

[19] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37(2):99–122, 2007.

[20] S. Schiaffino and A. Amandi. User - interface agent interaction: personalization issues. *International Journal of Human-Computer Studies*, 60(1):129 – 148, 2004.

[21] S. N. Schiaffino and A. Amandi. Polite personal agents. *IEEE Intelligent Systems*, 21(1):12–19, 2006.

[22] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[23] B. Shneiderman. Response time and display rate in human performance with computers. *ACM Comput. Surv.*, 16(3):265–285, 1984.

[24] D. S. Weld, C. Anderson, P. Domingos, O. Etzioni, K. Gajos, T. Lau, and S. Wolfman. Automatically personalizing user interfaces. In *In IJCAI03*, pages 1613–1619, 2003.

[25] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition, 2005.

[26] Zona Research Report. The need for speed, July 1999.