An Efficient Method to Reduce Startup Time of Applications

A Dissertation Submitted in Partial Fulfillment of the Requirement for the

Award of Degree of

MASTER OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

Submitted By

MOHIT KHANNA

Roll No. – 2K11/CSE/07

Under the Esteemed Guidance of

Dr. KAPIL SHARMA

Department of Computer Engineering

Delhi Technological University, Delhi

2012-2013

CERTIFICATE

DELHI TECHNOLOGICAL UNIVERSITY

BAWANA ROAD, DELHI – 110042

Date: _____

This is to certify that dissertation entitled "**An Efficient Method to Reduce Startup Time of Applications**" has been completed by Mohit Khanna in partial fulfillment of the requirement of major project of **Master of Technology** in **Computer Science and Engineering**

This is a record of his work carried out by him under my supervision and support during the academic session 2012 -2013.

(Dr. KAPIL SHARMA)

RESEARCH GUIDE

Dept. of Computer Engineering

Delhi Technological University, Delhi

ACKNOWLEDGEMENT

It gives me a great sense of pleasure to present the Thesis on my Research Work, undertaken during the course of M.Tech. I owe special debt of gratitude to my Supervisor and Guide ***Dr. Kapil Sharma, Associate Professor****, Department of Computer Science & Engineering, Delhi Technological University, for his continual support and guidance. His sincerity, thoroughness and perseverance have been a perpetual source of inspiration for completion of this research. It is because of his unflinching motivation and faith in my work and efforts that our endeavors have seen light of the day.*

I also take this opportunity to convey my deepest gratitude to ***Dr. Vijay Rao, Scientist 'F', Institute for Systems Studies & Analyses, DRDO, Delhi*** *without whose guidance, support and efforts the simulation part of our Research could never have been possible; and without the initial simulation results, we could never have realized the deep potential that our concept discussed in this Thesis actually holds.*

I would also like to sincerely acknowledge the kind assistance and cooperation of ***Dr. Richa Mishra and Dr. Rajni Jindal*** *who had been my initial guides during my M.Tech. Course and had always motivated me to strive forward and progress with this idea which is now presented in this Thesis.*

Last but not the least, I would like to thank God, my parents and friends for their blessings and best wishes which certainly seem to have worked in making this small seed of idea sowed a year and a half ago, into a strong tree with sweet fruits of appreciable results today.

(MOHIT KHANNA)

Master of Technology

(Computer Science and Engineering)

Department of Computer Engineering

Delhi Technological University, Delhi

ABSTRACT

An application usually takes certain time for its loading during which the executable file and the other related files are brought to the main memory and necessary initializations routines are run. However, it has commonly been observed that in an actual use of an application by a general user, several of the loaded components are not essentially used during the lifetime of its execution. This results only in an increased space taken by them in the memory as well as an increased burden during their loading into the memory in terms of CPU cycles and memory wasted and time taken to load and initialize them, thus increasing the overall startup time of the software. In this Thesis, we propose a methodology by which we track the user's usage pattern, and by allocating certain weights to some specific static and dynamic parameters of the software and its usage style, we selectively determine the potential components that are most likely to be used in the subsequent execution of that software and load only those necessary components at next startup. Thus, each time the user runs the software, he is presented with the set of components that he is most likely to use in that specific execution. We have done a simulation based on Monte Carlo approach and we have observed a reduction of around 40-50% in software load time and an approximately 70-80% Hit Rate for the components requested by user against the actually loaded components. We then developed a test application and compared its startup times and memory requirements at startup, and found around 59% reduction in application startup time and 16% reduction in start time memory requirements when our proposed method is used.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

Faster devices and faster applications have always been preferred by the people worldwide. Researchers have been trying hard since decades to develop techniques which assist in making the applications execute faster. One very important region in an application's execution which gives the first impression of an application's speed is the time it takes for its startup. The period of time between the command to initiate the program and the time at which the program commences processing to external events, depends upon both the time it takes to load the program into memory and the time that it takes to execute initialization code [1].

Appreciable work has been done at various levels to improve the startup times of applications. At hardware level, using hybrid drives or solid state drives has been one very effective method, but still developers have been struggling to improve the startup speed from their ends due to a never satisfying user expectation for faster applications. Although using optimized algorithms for the initialization routine in application code improves the performance appreciably, yet one lesser tapped area is to selectively load the application components.

If the application is developed in form of dynamically loadable components, then it is possible to reduce the disc access time as well as the time consumed in initialization routine by loading only those components that the user is expected to use. This would result in much improved startup times.

However, a big question arises that how to make this prediction before application startup that which components the user would use in that execution, so that only those components be loaded for that execution. We have proposed a methodology for that called K-Method which makes use of a formula that we have proposed called K-Formula that computes Load Influence Value for each component. Load Influence Value for a component determines the importance to be given to that particular component for being loaded at startup of the next execution.

1.1 MOTIVATION

Application startup time primarily involves two components – to bring the application code and related files into the memory which accounts to disk access time, and other being the time taken to execute the initialization routine.

However, most of the features offered by the application are not used by a common user. At the same time, no distinct subset of features exist that all the users would use, and this small proportion of the features that users use include different features for different users. Thus, if we selectively load only those components that provide the features that user wants to use, then this can lead to improvement in startup times by reducing disk access time as well as time required to run initialization code for the features that a user does not uses.

Currently, this selective component loading is being done manually, wherein the developer provides a particular set of components providing certain features as default, and the user can later change this list of features that he wants to be loaded in the application manually.

This however includes a static factor into the selective loading in terms that for several application executions, the same set of components would be loaded that are manually defined as 'can be used' by the developer or user. This means that still several components are being loaded which might not actually be used during application startup.

There is a need to automate this selective component loading technique so that the component set that has to be loaded get dynamically updated according to the pattern in which the user interacts with the application.

To this effect, we have proposed our methodology which performs the above said automation.

1.2 PROBLEM STATEMENT

In our research work, we have focused to improve the startup times of component based applications. This we achieve by predicting and selectively loading the components that user is expected to use during the application execution. For this we developed an automation technique which helps in making a choice of whether to include or not include a particular component at application startup.

The Problem Statement for our Thesis can be proposed as:

"To Develop an Efficient Method to Reduce Startup Time of Applications"

1.3 SCOPE OF WORK

Our research work revolves around creating a method for reducing startup time of applications. We do this by selectively loading the application components at startup. Making an efficient choice of components for loading selectively is very important so that most of the requests that user makes for features get satisfied by the components already loaded at startup.

We have developed a new method (K-Method) which uses our proposed formula (K-Formula) to compute the Load Influence Values for each component. Load Influence Value for any component relate to the importance which that component holds for being loaded at the application startup. K-Formula requires its three constants (collectively called Configuration Parameters) to be found optimally by simulation for each new application being developed. Simulation also shows us the expected proportion of components that will need to be loaded and the HIT% which indicates what proportion of user requests for different features are met by the components already loaded.

We further develop a test application and test it for its startup speeds when run using K-Method and when run in conventional manner, i.e. without using K-Method. The performance analyses support our concept and the results are very appreciable.

Broadly, the scope of our work can be summarized as:

- To develop a method for selectively loading of components with automation used for making a choice of which components to load. (K-Method)
- To device a formula that determines how important a particular component is for loading at the next execution. (K-Formula)
- To perform simulation for determining the optimal values of three of the constants used in K-Formula.
- To use simulation to determine the improvement possible in startup time.
- To develop a Test Application that works on K-Method
- To do performance analyses for startup time and memory consumption at startup, for the Test Application when used with K-Method and when run without K-Method.

1.4 ORGANIZATION OF THE THESIS

The remained of this Thesis is organized in form of the following chapters:

Chapter 2: Literature

This section gives an overview of the different techniques that have been used to improve startup times of the applications. These include the techniques used at the Hardware level, Network Level, Application Level, as well as Linker Load Time Optimization Level

Chapter 3: A Glimpse of Journey from Application Installation to Application Execution

This section briefly describes what all goes on in the background while an application is being installed and executed. This section further describes the different regions in the above tasks where optimizations can lead to improved startup time. Thereafter discussion is done of the yet lesser tapped area (selective loading of components) and insights are given for as to why this can be a good place to work for improving startup times.

Chapter 4: Our Idea for Improving Startup Times

In this section, we have described our proposed method (K-Method) and our proposed formula (K-Formula) which is used in the K-Method. Thereafter, simulation is performed and results of simulation are discussed.

Chapter 5: Test Application Based on K-Method – 'Advance Browser'

This section describes the Test Application that we have developed for testing our method on practical grounds. Performance Analyses is done on the Test Application for cases when it uses our proposed method and when it does not. Results suggest drastic improvement in startup time when our proposed method is used.

Chapter 6: Conclusion

The conclusion of our research work is presented in this section.

Chapter 7: Future Work

This section discusses about the other domains where the research presented in our Thesis can be expanded to.

Chapter 8: References

This section gives a list of citations used in our thesis and research work

II. LITERATURE

Researchers have been working since decades in an attempt to make programs start up faster. Various optimization techniques have been applied primarily at the Hardware Level, Network Level, Linker Level, as well as at the Application Level, as shown below in Figure 1:

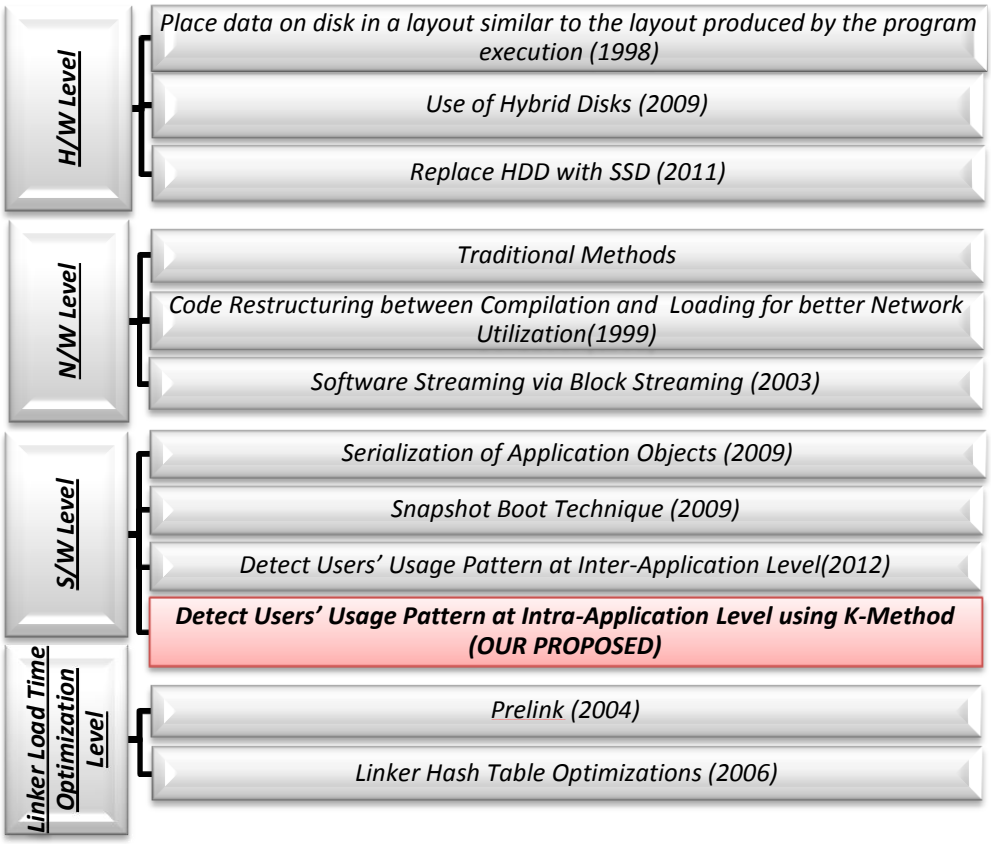


Figure 1: Different Techniques for Speeding up Application Startup

2.1 AT HARDWARE LEVEL

1. Rearranging Disk Data in Layout Similar to that produced during Program Execution

I/O activities consume approximately 85% of the time during the loading of an application. According to Amdahl's Law [2] , the overall application load time is limited

by the slow disk I/O activities. Yin proposed a method which increased speed of disk operations by reducing the seek time [3]. Other works [4], [5], [6] showed that it is possible to improve disk performance by reorganizing disk data, but they didn't attempt to determine an optimal layout for a deterministic access pattern. They reorganized the disk based on the idea that a disk's performance can be improved by clustering frequently accessed data. Yin suggested using simulated annealing, genetic algorithms and other algorithms to find a nearly optimal data placement which resulted in around 20-30% improvement in startup time of applications [3].

On similar logic, Walsh described his patented method which makes use of 'Sequence Lists' to determine the order of files' placements on disk during the installation time itself [7]. After build of a program is completed, the program is launched and the disk activity associated with disk-intensive operations is monitored to determine the order in which file portions are read from a disk during program or command launch. This data is used to create a load sequence list, which indicates the order in which various portions of the files are read during launch. The installation disks include the files and the load sequence list. During the installation process, the installation program reads the data from the load sequence list and writes the file portions so they are stored in the order prescribed by the load sequence list in contiguous clusters on the hard disk drive. The computer can then read launch-related data from the disk in the proper order from contiguous disk clusters, which minimizes or eliminates wasted time that would have resulted from disk accesses if the disk heads had to move between non-contiguous clusters in order to read the launch-related data.

2. Use of Hybrid Disks

We are aware that application launch times, which are important to users, are primarily bounded by disk seek times. A solid-state disk has a negligible seek time, but large solid-state disks are not cost-effective. A hybrid disk, consisting of a large disk drive and a flash memory of smaller capacity, can provide a reasonable compromise [8].

SuperFetchTM is an advanced prefetching technique, supported by Windows Vista, that exploits the flash memory of the HHDDs or the free space in the main memory in

absence of flash [9]. It analyzes the memory utilization patterns of a user and then prefetches the data from the HDDto the main memory, which will result in significant reduction in the launch time. Flash memory cards or the flash memory in HHDDs are managed by Windows ReadyBoostTM and Windows ReadyDriveTM, respectively [9]. However, the performance of SuperFetchTM is critically dependent on its hit ratio, and thus it requires a flash memory which is at least as large as the main memory, and which may be two or three times as big, to achieve appreciable performance benefits. Such a large flash memory may significantly increase cost of a system.

Moreover, there is no systematic approach to the allocation of portions of launch sequences to solid-state memory to achieve the shortest application launch time. As a solution, Joo showed how to reduce application launch times with a hybrid disk by pinning only a small portion of an application launch sequence into flash memory [8]. He modeled the latency of a hybrid disk, analyzed the behavior of application launch sequences, and formulated the choice of the optimal pinned set as an integer linear programming (ILP) problem. Experiments showed that this approach reduced application launch times by 15% and 24% on average, while pinning between 5% and 10% of the application launch sequences into flash memory.

3. Replacement of HDD with SSD

One of the most effective ways to improve application launch performance is to replace a hard disk drive (HDD) with a solid state drive (SSD), which has recently become affordable and popular. A SSD consists of a number of NAND flash memory modules, and does not use any mechanical parts, unlike disk heads and arms of a conventional HDD. While the HDD access latency—which is the sum of seek and rotational latencies—ranges up to a few tens of milliseconds (ms), depending on the seek distance, the SSD shows a rather uniform access latency of about a few hundred microseconds (*u*s) [10].

Joo has proposed in his paper, a new application prefetching method, called the *Fast Application STarter* (FAST), to improve application launch time on SSDs [10]. The key idea of FAST is to overlap the computation (CPU) time with the SSD access (I/O) time

during each application launch. To achieve this, the sequence of block requests in each application is monitored, and the application is simultaneously launched with a prefetcher that generates I/O requests according to the *a priori* monitored application's I/O request sequence. FAST consists of a set of user-level components, a system-call wrapper, and system debugging tools provided by the Linux OS. FAST can be easily deployed in most recent Linux versions without kernel recompilation. FAST has been implemented and evaluated on a desktop PC with a SSD running Linux 2.6.32, demonstrating an average of 28% reduction of application launch time as compared to PC without a prefetcher.

Apart from this, a technique called Application XIP was suggested in another paper which would by-pass the RAM itself [11]. When a program is executed, the kernel program loader maps the text segments for applications directly from the flash memory of the file system. This saves the time required to load these segments into system RAM.

2.2 AT NETWORK LEVEL

Systems based on mobile code, such as virtual machines like Java [12], Inferno [13] and OmniWare [14], and embedded object systems like ActiveX [15], inherently require that clients fetch applications over the network prior to their execution. Thus the startup time also includes the application transfer time. Several techniques have been given to reduce the application transfer time which are listed below:

1. Traditional Methods

Previous work on decreasing application transfer times broadly spans traditional compiler optimizations, code compression, overlapped I/O and lazy loading. Traditional compiler techniques for reducing the static size of applications have centered on instruction selection. Such optimizations often take place in the back end of a compiler, where the emitted instructions are selected to reduce the memory footprint of the instruction segment using standard techniques [16], [17]. Code compression is another field that examines the effective bandwidth of application transfers. Code compression relies on a post compilation step to represent applications using a space-efficient encoding. Krintz

proposed modifying Java's execution semantics in order to overlap code transmission with execution [18]. Such an approach offers increased application performance by allowing I/O operations of a thread to be scheduled concurrently with the computation of the same thread. The authors report through simulation results that this scheme can achieve 25-40% performance improvement. In another paper, Lee propose combining code reordering and demand paging to improve the startup of ActiveX applications [19]. He showed that code reordering via binary rewriting and demand fetching of pages through the memory fault handler can improve the startup times by up to 58%.

2. Inserting a Separate Code Restructuring Step between Compilation and Loading to better Utilize Network Bandwidth and Improve startup time

It can be observed that for network based applications, compliance with existing design and coding standards and minimizing the impact on the clients is crucial [20]. Fundamental problem for mobile code is that the units of code distribution in networked object systems, such as Java, are not suited for efficient utilization of network bottlenecks. To help solve these problems, Sirer proposed adding up a separate optimization step between compilation and loading, whereby application code is split up into smaller transfer units based on a profile to more effectively use the available network bandwidth for program download [20]. This approach uses binary rewriting to repartition application components at method granularity such that the frequently used related code units are grouped together, while less frequently used methods are factored out into chunks that can be transferred separately and independently. This repartitioning is performed late in the software distribution chain, after the code has been released but before it has been shipped to the users. The server uses profiling to discover common application paths, and repartitions the application through binary rewriting to make these paths execute faster. This approach provides a practical way for profiling and repartitioning in the context of the Java virtual machine that does not require any modifications to the clients, and yet achieves up to 30% reductions in startup time for interactive applications.

3. Software Streaming via Block Streaming

Software streaming allows the execution of stream enabled software on a device even while the transmission/ streaming may still be in progress. Thus, the software can be executed while it is being streamed instead of causing the user to wait for the completion of download, decompression, installation and reconfiguration. In his paper on this work, Kuacharoen proposes a streaming method called Blok Streaming which can reduce application load time as seen by the user since the application can start running as soon as the first executable unit is loaded into the memory [21].

However, the software to be streamed must be modified before it can be streamed over a transmission media. After normal compilation, the software must be partitioned into parts for streaming through a process streaming *software streaming code generation*. After modification, the application is ready to be executed after the first executable software unit is loaded into the memory of the device. In contrast to downloading the whole program, software streaming can improve application load time. While the application is running, additional parts of the stream enabled software can be downloaded in the background or on-demand. If the needed part is not in the memory, the needed part must be transmitted in order to continue executing the application. The application load time can be adjusted by varying the size of the first block while considering the time for downloading the next block. This can potentially avoid the application suspension due to block misses. The predictability of the software execution once it starts can be improved by *software profiling* which determines the transmission order of the blocks. The application load time for the sample application improved by a factor of more than 10X when compared to downloading the entire application before running it.

2.3 LINKER LOAD TIME OPTIMIZATION LEVEL

1. Concept of Prelink

Prelink is a popular tool used to decrease program load time, shorten system boot time and to make applications start faster. It relocates libraries on disk to save dynamic linking time [22].

When the dynamic linker loads a dynamically linked ELF binary, it has to also load and link all of the libraries before executing the program's entry point, _main(). This process involves relocating libraries—changing all addresses referenced in the library to reflect the actual addresses in memory. Relocating libraries involve iterating through each address in the library and replacing it with the real address as determined by the library's location in the process's virtual address space. The relocation process will slow down an application's launch. In order to speed up the process, Prelink relocates the libraries ahead of time. This is done by scanning every executable to be prelinked, generating a graph of libraries that will be loaded at the same time as other libraries, and then calculating target addresses for each library such that it will never be loaded at the same address as other libraries. These offsets are then stored in the shared object files themselves, and the symbol tables and segment addresses are all adjusted to reflect addresses based on the chosen base address. This method led to a speedup of OpenOffice.org 1.1 by 1.8s from 5.5s on 651MHz Pentium III [22].

2. Hash Table Modifications

Reduction in dynamic linking time by linker hash table modifications has resulted substantial gains in application load time [23].

In a typical operation, symbols are stored in hash tables in ELF binaries which are kept small, and symbols that hash to the same value are compared by a simple string comparison. However, symbols in the same bucket with the same prefix need a long string comparison which is a slow task. Drepper proposed to utilize a GNU linker optimization that focuses more on producing short hash chains than a small hash table size [24]. The shortened length of hash chains mean that symbol look-ups do not have to perform as many string comparisons. This linker optimization can be activated by passing -Wl,-O1 to gcc at link time.

Meeks proposed that by passing -Bdirect at link time, the build process can cause many symbols to be directly linked allowing the dynamic linker to severely decrease the search space during lookup [25].

In another optimization step by Meeks, the dynsort keyword is added to the GNU linker. By passing -zdynsort at link time, the .dynsym and .dynstr sections as well as relocations are all sorted by ELF hash and by the position where they land in the hash bucket. When symbols are looked up in an ELF object, a hash table has to be searched. With dynsort, the symbols that have to be examined while walking a bucket in hash table are all adjacent to each other. This reduces the number of L1 and L2 cache misses, allowing the CPU to utilize its facilities much more efficiently during dynamic linking [25].

2.4 AT APPLICATION LEVEL

1. Serialization of Application Objects

The period of time between the command to initiate the program and the time at which the program commences processing to external events, depends upon both the time it takes to load the program into memory and the time that it takes to execute initialization code [1]. Any technique that reduces the amount of time spent executing initialization code will decrease the amount of time until the application process external events

To this effect, Wolff has discussed his patented method wherein application speed is improved by providing a serialized representation of application objects to the runtime environment [1].

In one aspect, the invention is a method of developing a faster loading application. It includes steps of compiling first object code for an application; loading the application into a first runtime environment; creating a serialized representation of a memory space in said first runtime environment; building second object code using said serialized representation; and deploying said second object code

In another embodiment the method may include the steps of compiling an application provided in a source language, initializing the application in a runtime environment, and creating a serialized representation of the application. Thus when the runtime finds the application objects present in a serialized fashion, then it leads to lesser cache miss and page faults and thus an increase in loading speed is observed.

Jo improved the startup latency of a commercial digital TV by 35% by following a better initialization order and determining when and which data should be prefetched to the buffer cache [26].

2. Snapshot Boot Technique

Kaminaga proposed a snapshot boot mechanism to enhance the startup time of a commodity OS by using the suspend-resume technique [27]. The snapshot boot technique suspends the running kernel, and resumes with the suspend image instead of booting. To reduce resume time, it resumes the suspend image from a boot loader, and not the kernel as in general Suspend-Resume (Hibernate) technique. With the snapshot boot technique, he also applied other techniques such as XIP and pre-linking. However the biggest problem with snapshot boot is that it takes a long time to create images and then save them to the storage device during the suspend process, because it targets every page. In addition, if the switch is turned off while creating an image, then the image is not created, and the resume process cannot be executed.

Inwhee proposed a method to improve upon these problems [28]. With his method, the system does not generate a snapshot image at the end of the system process; rather, it generates one snapshot image for the first bootup so as to improve snapshot boot problems. The generated snapshot image is then used for every bootup to reduce the time needed for the existing snapshot boot. Thus overall boot performance improved even for cases where system fails to generate and save the snapshot image. This eventually led to an improvement in application startup time.

3. Detect Users' Usage Pattern for Different Applications

Preload, proposed by Esfahbod, is an adaptive daemon that prefetches binaries and shared libraries from the hard disk to main memory on desktop computer systems by monitoring the applications that the user runs [29]. It is based on a Markov-based probabilistic model capturing the correlation between every two applications on the system. The model is

then used to infer the probability with which each application may be started in the near future. These probabilities are used to choose binaries and dependencies to prefetch into the main memory. An improvement of around 25% has been observed for smaller desktop applications and around 50-55% for larger desktop applications.

On similar grounds, Hokwon proposed a Usage Pattern-based Prefetching scheme, called UPP, which is suitable for mobile devices [30]. To inspect the usage patterns, the dataset of the application usage is collected and analyzed. Additionally, considering mobile devices which have relatively poor hardware resources, the lightweight prediction model is employed. Using UPP, the startup time got improved by about 5%, and the accuracy of the prediction got shown up to 59% for the practical dataset.

However, both these methods work by tracking the pattern in which the user uses different available applications. Prediction is made for the next application that user can request and its related files are brought to the memory beforehand.

However, in our research, rather than tracking pattern at in which use uses different applications (inter-application level), we track the pattern in which user requests for the different within a specific application (intra-application level). We then predict which components could be used in the next execution of the application and when the application is next started, we selectively load (or initialize) only the predicted components, making the application adaptable to user's usage pattern over a period of time, resulting in appreciable improvement in application load time.

III. A GLIMPSE OF JOURNEY FROM APPLICATION INSTALLATION TO APPLICATION EXECUTION

A whole bunch of several activities run in the background when a user clicks on the application executable icon or gives a command manually to execute the application from terminal in Linux or command prompt in Windows. However, these background tasks consume considerable time and this appears as an unwanted delay in the application startup to the user. Our research focuses on reducing this delay that user experiences in starting up of an application. However, an understanding of the basic steps that occur during application installation and thereby its execution is crucial to determine what all possible measures can be taken to improve the startup speed.

3.1 BACKGROUNDS OF APPLICATION INSTALL PROCESS

A computer system contains several applications stored in its hard disks. Throughout its lifetime, the user installs and uninstalls many different applications at different times. During Installation several complex tasks occur in the background. Once the application package has been uncompressed, system requirements are checked to ensure whether the application can be safely installed or not. Existence of previous versions is checked to determine whether an upgrade is required or a clean install is required.

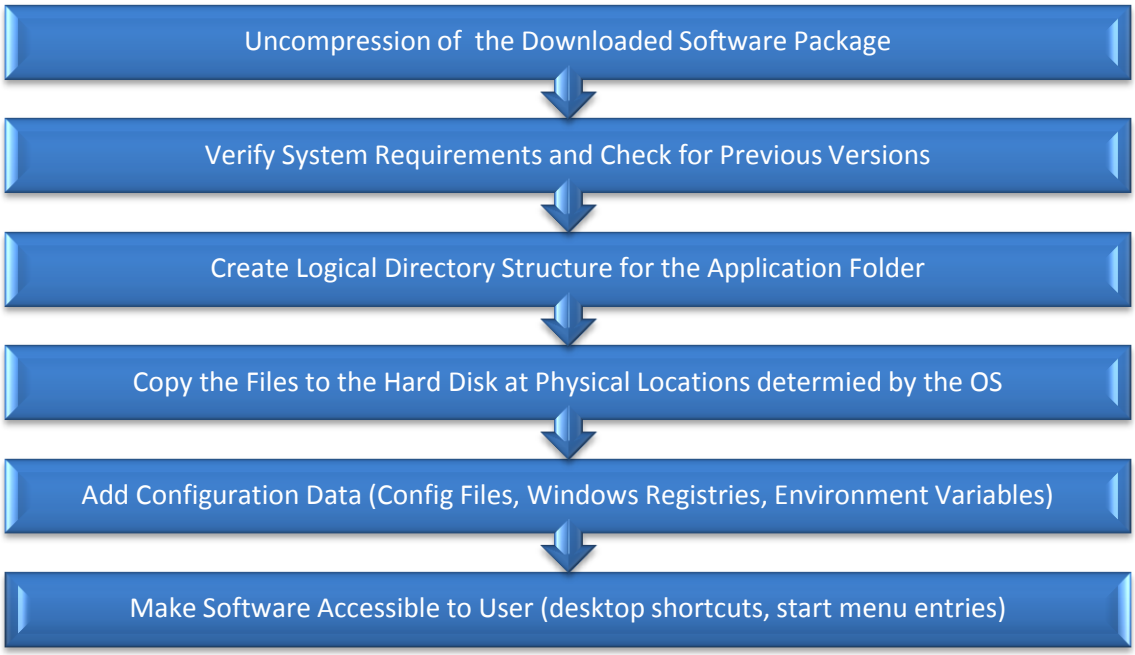


Figure 2: Steps in Application Install Process

Logical Directory Structure for the application is created at a dedicated or a user specified location and the relevant files are copied to the hard disk in an order and physical location as determined by the Operating System. An interesting fact here is that the different files of the same application may not necessarily be contagiously stored. Further, a large file may also be chunked and stored separately if so determined by the Operating System. Apart from physically writing the application files to the hard disk, configuration data such as configuration files, registries (for Windows OS) and environment variables are added as well. Thereafter links, bookmarks, shortcuts on desktop, start menu entries, and other such accessibility options are created for the user to be able to run the application when desired.

3.2 BACKGROUNDS OF APPLICATION EXECUTION PROCESS

When a user clicks on the shortcut icon for executing the application, a command is sent to the Operating System which determines the physical location on the hard disk where the executable file of the particular application is actually stored. The executable file and other related files are then read and the code and the related data is brought into the main memory where the executable is prepared for running. This task is performed by an Operating System Routine called Loader. A Process is created in the Main Memory for the application and the Process Control Block (PCB) is created in the Operating System Kernel and associated with this newly created process. Once the process is ready, its PCB is allocated to the ready list to be picked up by the Dispatcher which allocates it to the Processor for the execution. This execution further leads to performing certain start time initializations of supporting components or features. It is after all these background steps that the user finally sees a useful screen at the application workspace and his wait for the application startup delay for the current execution gets over.

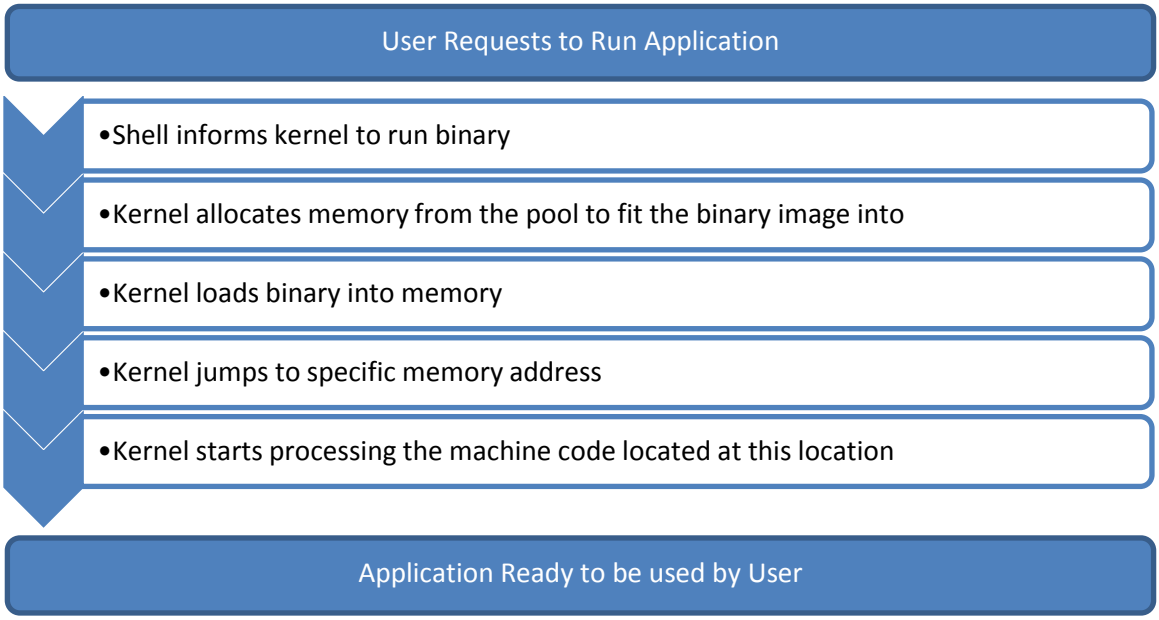


Figure 3: Steps in General Application Execution

Loading of supporting libraries may also be done dynamically at Run Time with help of a dynamic loader. This mechanism allows the application to startup in the absence of these libraries, to discover available libraries, and to potentially gain additional functionality. [31] [32]. For example, in browsers like Mozilla Firefox, this includes initializations of several user specified extensions, plugins and add-ons.

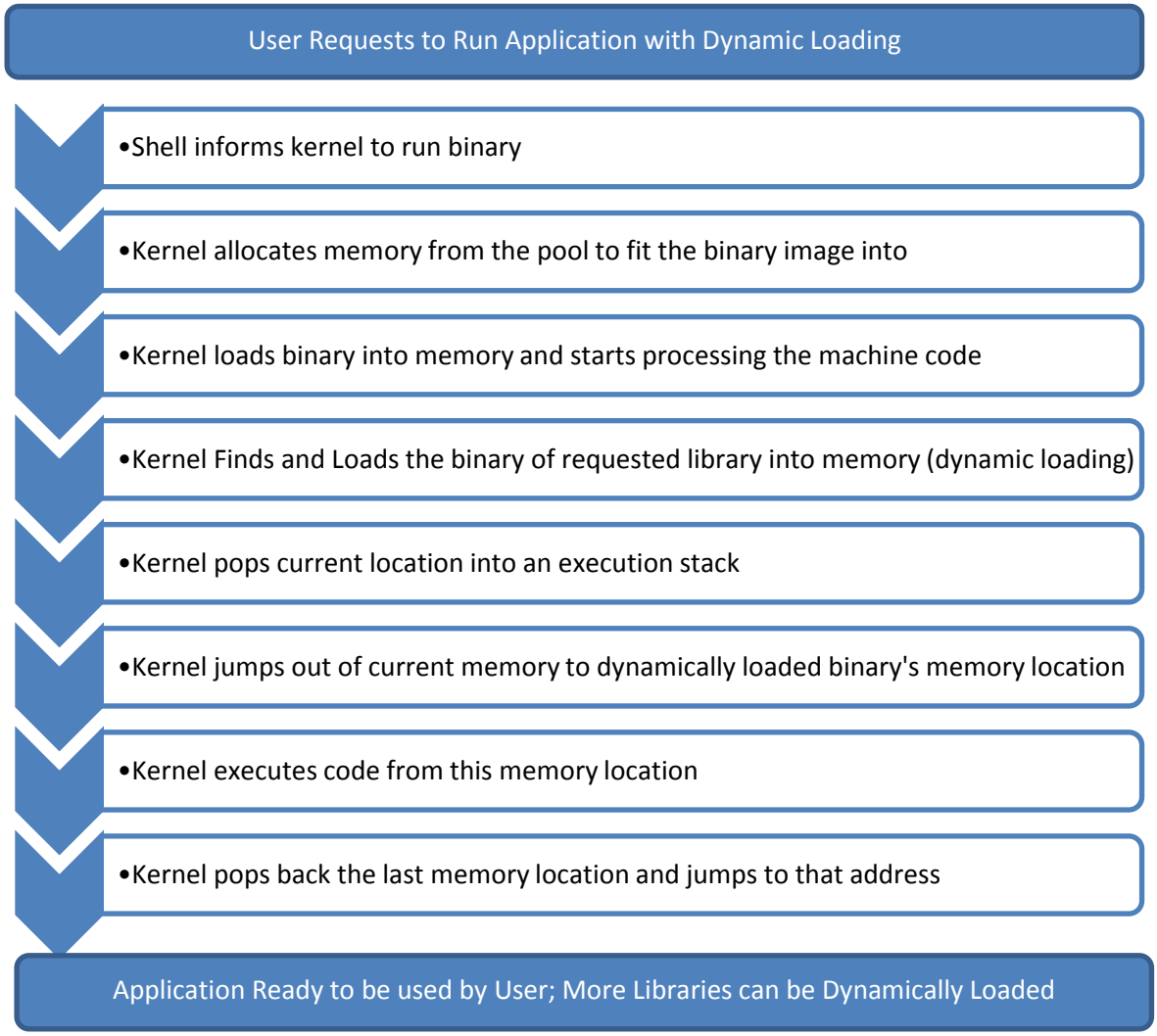


Figure 4: Steps in Application Execution for an Application with Dynamically Loadable Libraries

3.3 WAYS TO IMPORVE STARTUP SPEED OF APPLICATIONS

For examining the performance characteristics of a workload, IBM has proposed a dynamic model of program execution [33]. The same can be used for determining the locations where improvisations can lead to improved startup time.

Program Execution Hierarchy. As shown is Figure 5, Process Execution Hierarchy can be shown as a triangle on its base. The left side represents hardware entities that are matched to the appropriate operating system entity on the right side. A program must go from the lowest level of being stored on disk, to the highest level being the processor running program instructions. For instance, from bottom to top, the disk hardware entity holds executable programs; real memory holds waiting operating system threads and interrupts handlers; the translation lookaside buffer holds dispatchable threads; cache contains the currently dispatched thread and the processor pipeline and registers contain the current instruction [33].

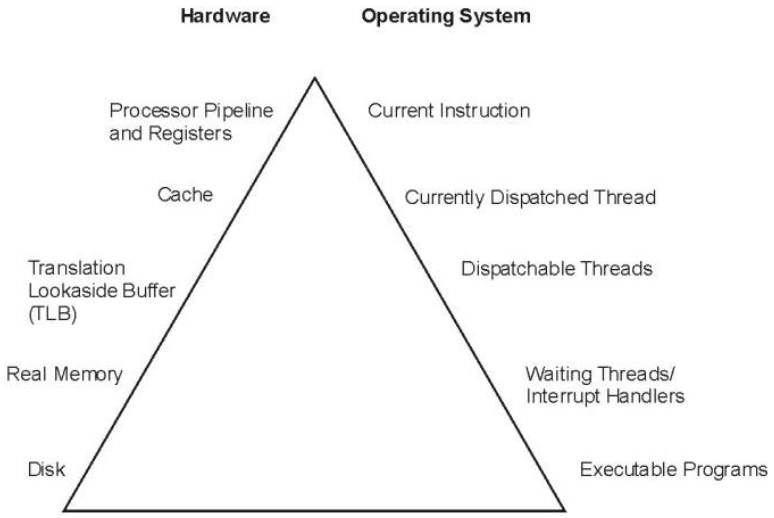


Figure 5: IBM's Process Execution Hierarchy [33]

To run, a program must make its way up both the hardware and operating-system hierarchies in parallel. Each element in the hardware hierarchy is more scarce and more expensive than the element below it. Not only does the program have to contend with other programs for each resource, the transition from one level to the next takes time. Thus, the startup time gets limited by the transition time required for moving from one level to another.

Starting from the base of hierarchy, the time required for bringing the program code from the hard disk to the main memory is the largest. This is due to the fact that this involves physical movements of the disk (rotation) and the disk head (seeking). Thus an improvement in the binary access time would lead to improvement in overall application startup time. This can be done in several ways. One obvious way is to make use of faster hard disks with more rotation and seek speed, or to use disks with multiple read write head and platters. Another technique makes use of the fact that data for the same application is stored in a scattered manner across the disk. A reordering is done of the files on the hard disk in such a manner that all the files required during application startup come together in the same or adjacent sectors in an order similar to that in which they would be required during startup. This drastically improves the startup speed as now a much lesser head movement would be required. Another method is to make use of solid state drives instead of the hard disk drives. Since solid state drives require no physical movements, hence they have a much faster data access rate than hard disk drives.

Further in the hierarchy, when some required file is not found in main memory it leads to a miss and the file is fetched from the hard disk leading to a considerable delay in form of the disk access time. So, one intuitive way to improve startup time is to increase the main memory of the system and hence the page table size which would lead to reduction in the number of miss. Another method is to apply techniques to increase the probability of finding the required file in the main memory. This is done by techniques like preloading and snapshot booting. In preloading the applications to be loaded in near future are predicted and then their corresponding files are brought into the main memory even before any command for its execution is fired. By the time user executes the application, the required files are already in the main memory, and thus the penalty on retrieving data from hard disk is prevented. In snapshot boot technique, the state of the executing processes is stored and saved onto hard disk. The next time system in switched on, the stored snapshot image is transferred back to the main memory. Another method is to observe which all applications a user would execute next, but monitoring his usage pattern over a period of time. At cache level, using larger cache size will improve the startup speed. At linker level, improvisations can be made by linking the libraries ahead of time and thus saving the linking time during program startup.

3.3.1 A YET LESSER TAPPED AREA – PARTIAL LOADING

Although appreciable improvements have been achieved in startup times by the techniques discussed earlier, however the human nature to have faster loading and executing applications leaves the software companies in putting in huge investments in terms of cost and money to optimize their applications to make startup faster. This can be understood by analyzing the very common fact that a high competition has always prevailed among the browsers like Internet Explorer, Mozilla Firefox, Google Chrome and Opera for improving user experience by starting up the browser application faster and loading the homepage quicker.

Though techniques like using prefetching, or using optimized algorithms in code do help, but one intuitive but yet less tapped technique is to load the application only partially. Instead of bringing the entire application and its dependencies into the memory all together, it would be much better to predict what features the user would be using and then bring only that part of the application to the main memory, and execute. This selective feature loading and runtime binary linking requires that the application be developed in a plugin or component based architecture. The different features or components that the user can use, should be available separately in form of add-ons, extensions or plugins and dynamic loader should be able to link the dynamic components at run time when requested.

3.3.2 MOTIVATION BEHIND THE PARTIAL LOADING TECHNIQUE

A research in year 2000 was done to analyze how many features of a Bloated Application are familiar to a general user and with how much frequency are they used. On surveying 53 participants for their use of Office 97 comprising 265 features, following results were observed [34]:

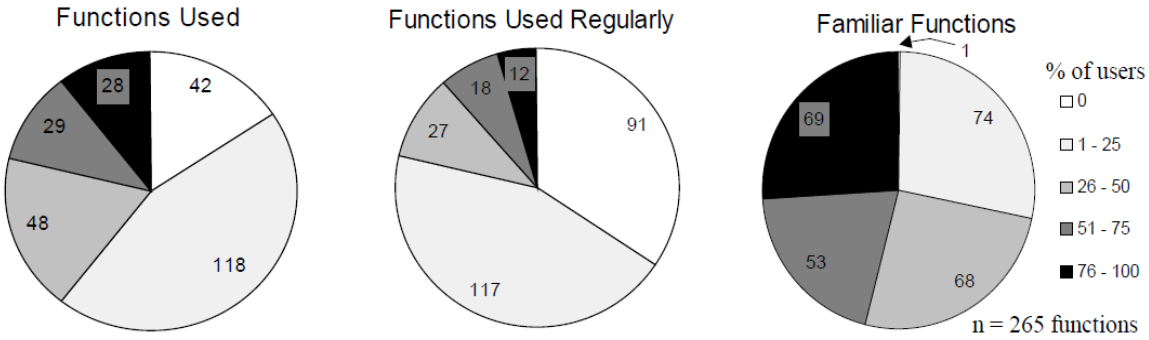


Figure 6: Number of functions that are used, used regularly, and are familiar to users [34]

Figure 6 shows a clear indication of the fact that most of the users are familiar with a very small subset of the total features offered by the application, and a general user uses even much lesser number of features than what he is familiar with. Further, looking at the relationship between familiarity and use from the perspective of the individual user it can be that a relatively low percentage of the functions are actually used; on average participants are familiar with 51%, and use 27% of the functions. [34]

	First-level functions
Average # of functions familiar to participants	135 (51%)
Average # of functions used by participants	72 (27%)
Average # used regularly	40 (15%)
Average # used irregularly	32 (12%)
Maximum # familiar to any participant	245 (92%)
Minimum # familiar to any participant	24 (9%)
Maximum # used by any participant	119 (45%)
Minimum # used by any participant	8 (3%)

Table 1: Means and Ranges of Familiar and Used Functions (n=53) [34]

These results are in acceptance with the 90/10 rule which states that an average user spends 90% of his time in using just 10% of the product. [35]

The results show a possibility of improvement of startup speed by having only the features that the user uses. This means having a smaller installation package, lesser installation time, lesser number of files required to be brought to the main memory during startup and hence a faster startup. However, a different subset of features is used by most of the users which depends upon the type of work they do on the application and the style in which they do that. Further, research shows only 13 (25%) want to have unused functions removed entirely but 24 (45%) would prefer to have unused functions tucked away [34].

	Agree	**No Opinion**	**Disagree**
I am overwhelmed by how much "stuff" there is.	14	20	17
I want only the functions I use.	13	5	35
I prefer to have unused functions tucked away.	24	8	21

Table 2: Perception of Number of Functions on the Interface (n=53) [34]

This suggests for a need to allow the features to exist in the package but allow the user to choose which features he wants to install and also to choose which features he would like to have loaded on application startup.

3.3.3 CURRENT APPROACHES FOLLOWED FOR PARTIAL LOADING

In highly bloated applications like the Microsoft Visual Studio user is presented with dialog box to choose which features/ plugins/ components to leave out from the installation process [36]. This leads to user customized installation of the application with user selected features only. Once installed, some applications like Code::Blocks allow user to choose one among the several 'Perspectives' for that application usage. By choosing the default Perspective, a predefined set of features specific to that perspective only are loaded at application startup [37]. In applications like Mozilla Firefox and Google Chrome user has an option to manually disable the extensions, add-ons and plugins that he feels he won't require in near future [38] [39]. Mozilla Firefox also allows the users to make 'Profiles' which indicate the individual settings for each user [38]. A user profile having 20 add-ons and 5 open home tabs will have a much higher startup time than a user profile with 5 add-ons and 1 open home tab. Some applications like Microsoft Word and PowerPoint allow the user to customize the add-ons and modify the default ribbon views and functionality [40]. All these different techniques allow the applications to become more user specific and enhance user experience. Additionally, the number of features required to be loaded at start time is much lesser than the complete feature set. Hence, startup time reduces appreciably as well much lesser memory is required for the application's execution.

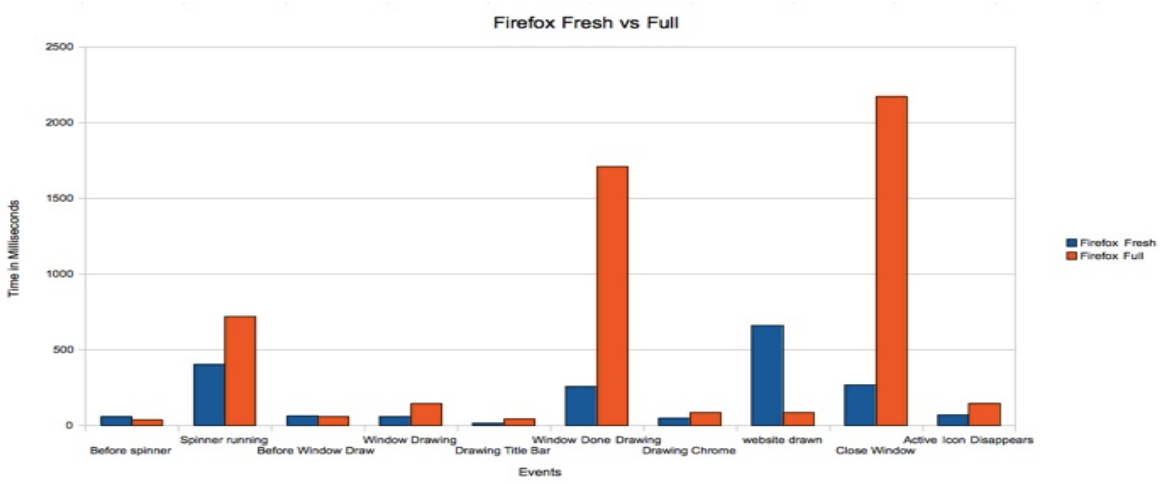


Figure 7: Comparison of Startup Time of Firefox Fresh v/s Full at Different Stages [41]

As evident from the above graphical results in Figure 7, given by a Mozilla Intern [41], time taken for a full feature startup (standard set of plugins; 50 bookmarks; 5 tabs in the session; 2 common add-ons) is much more than the time taken for a fresh feature startup (brand new profile; standard set of plugins enabled)

However, in all the techniques described above, the feature set selected to be loaded is either predefined by the developer, or manually customized by the user. To our knowledge, at present, no software makes use of a user pattern based dynamic choice in selecting the features to be loaded in next execution of an application with an intention to improve startup time. Our research focuses on developing a methodology by which the application usage pattern can be tracked and analyzed and then the components or features to be loaded for the next execution instance are determined.

IV OUR IDEA FOR IMPROVING STARTUP TIMES

4.1 BASIC IDEA – K-METHOD

Our idea, K-Method (*Khanna-Method*), for improving the startup speed of applications, involves monitoring the user's application usage pattern over time, finding out using K-Formula (*Khanna-Formula*) what all components are most commonly used by the user and loading only these useful components during startup. As discussed in previous section, most of the features of an application are not even known to the user and a much lesser proportion of the known features is actually used by the user. By monitoring the application usage pattern and hence on knowing which all features a user usually uses, we can find the feature set comprising of the most useful features from the entire set of all the features. On the subsequent application startups, our partially loaded application would load components or plugins for only these user specific features, thus drastically improving upon the startup time. As discussed earlier, the partial loading technique for almost all of the large applications involve loading of the feature set that was either predefined by the developer, or chosen by the user. However, these techniques suffer a major drawback that such feature sets are static in nature until manually changed next time by the user. Our idea overcomes this problem as our technique involves adaption of the feature set according to the application usage pattern of the user over a period of time. Thus, with time our application becomes more user-specific and more accurate in providing the user with only those features that he really needs and hence saving up memory as well as improving startup time which together makes the application execution appear faster. In case the user wants to use a feature that was not loaded during application startup, then the component/plugin providing that feature can always be loaded dynamically.

4.2 NEED FOR TWO TYPE OF EXECUTABLES

We have focused on identifying the components that may be required by the user in his next run of the software. However, we identify these components for two cases:

CASE 1: The software is being used after a considerably long time gap from its previous execution like more than 30 days or some other developer specific time constraint

CASE 2: The software is being used after a very short time gap from its previous execution.

For each case, we compute a *'Load Influence' (L.I.)* value for each component which determines the influence or importance that the component holds for being loaded the next time. Only the components having their Load Influence value higher than a dynamically computed threshold will be considered as having a high usage probability in subsequent run.

In Case 1, the components that are more generally used by the user will be loaded (i.e.) If the user is using the software after a long period of time, we expect that he would be doing the work that he has mostly being doing. For example, in spread sheet software, he might always have been using spread sheets only to record basic data and seldom uses the 'Create Formula', 'Compute Function' options, etc. So, these seldom used components

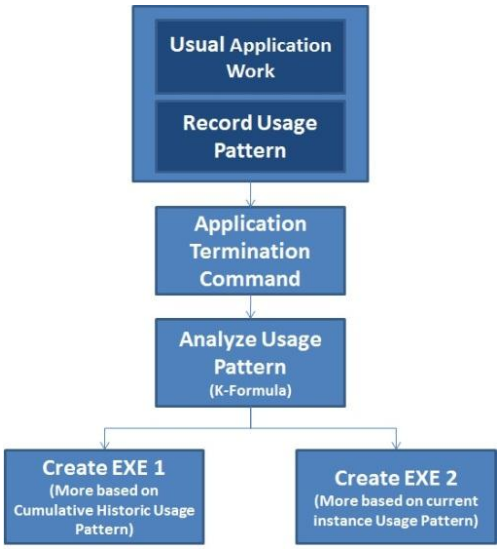


Figure 8: Concept of using Two Adapted Executables

will be assigned lesser preference. However, if he requests for one of these component that was not initially loaded, then it would be loaded dynamically.

In Case 2, the most recently used components are given a higher preference. This will be required in a scenario when a user is working on a project and may restart the software several times in short durations until his project gets completed. For example, Taking short breaks in between the project or continuing the same work over a period of several days.

In each instance, he would be requiring mostly the same set of components that are specific to that project. For instance, for the above same user, the subsequent run of the spread sheet software will this time also include the 'Create Formula' and 'Compute Function' components. This is because we can safely assume that if the same user is returning to the software within a very short span of time, then he is continuing with his previous work and would need the components requested earlier.

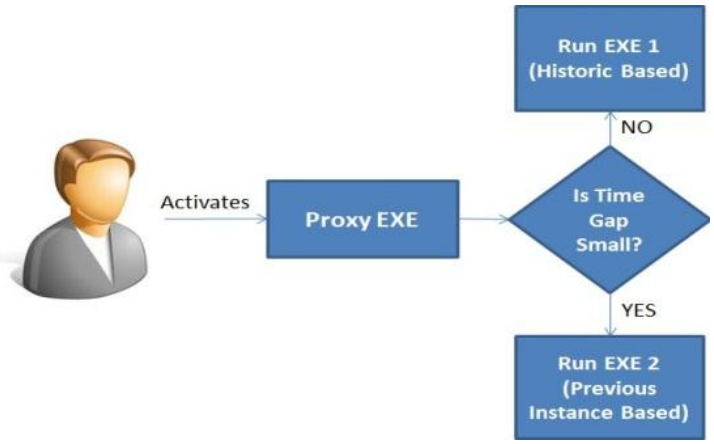


Figure 9: Choosing the Correct Executable at Startup

Once the components to be loaded the next time have been identified, we create two adapted executable files, one for each case and associate a timestamp with both of them.

For running the software, the user activates a 'proxy executable' which, based upon the difference between current time and the timestamp associated with executable, determines which the correct executable to be activated is.

An Alternative Approach

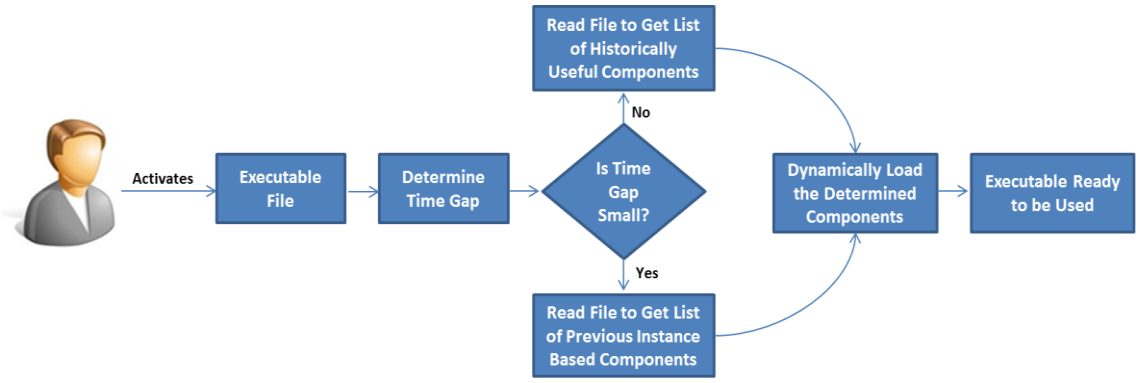


Figure 10: Alternative Approach for Loading Correct Set of Components at Startup

As an alternative to creating two separate executable, another approach is to write at time of software termination a simple file containing which components are required in each of the two executables. When a user activates the executable file during next run, only the components mentioned in the file are dynamically loaded based on time gap. So, effectively only one executable will be required and the time required to create two separate executables at program termination would be replaced by time required to create a simple text file which will be a considerable improvement

4.3 THE BIG QUESTION- HOW EXACTLY TO CHOOSE THE COMPONENTS FOR LOADING NEXT TIME

In this section we will be discussing the K-Formula which has been designed to compute the *Load Influence (LI)* of a particular component. Load Influence determines the importance that a component holds to be loaded in the next instance run. Higher the value of Load Influence for a particular component more is its probability to be loaded during the next instance of application startup.

4.3.1 Computation of Load Influence Values by K-Formula

Let there be a total of 'n' number of components from n_1, n_2, n_3 ... n_n. Amongst these, a subset of components is required to be determined which are most likely to be used in the next execution of the software. Whether or not a particular component would be used, and hence the *Load Influence* that each component would hold would be dependent upon the following application design time based and application execution time based factors:

Design Time Parameters

1. Alpha (α_i) - This is an importance factor given by the developer during the design time to component C_i. This may vary between 1 to 10. This is important to increase the probability of loading of those components which may explicitly be used less by user, but are otherwise important for the efficient functioning of the application, like a spell-check component. If there is no such specific importance to be given then keep alpha as zero.

   i.e. $LI_i \propto \alpha_i$

2. N_i - Number of components linking to the current component C_i.

   i.e. $LI_i \propto N_i$

3. K_i - Number of components linking from the current component C_i.

   i.e. $LI_i \propto K_i$

More the value of Ni and Ki, more is the coupling associated with the component, and hence more is the importance associated with the module to be loaded

Thus, based on the coupling associated with a component, its load influence can be shown as:

LI $\propto$ total incoming and outgoing links for that component/ total incoming and outgoing links for each component

i.e. $$LI_i \propto \left(\frac{(N_i + K_i)}{\sum_{i=1}^{tot\ Comp} N_i + \sum_{i=1}^{tot\ Comp} K_i} \right)$$

Further, adding up the developer specific importance factor Alpha, we get:

$$LI_i \propto \left(\alpha_i + \frac{(N_i + K_i)}{\sum_{i=1}^{tot\ Comp} N_i + \sum_{i=1}^{tot\ Comp} K_i} \right)$$

This part of the equation defines the importance that a component would hold due to its design time factors. Hence this can be considered as the Staticness in a Component's Load Time Influence.

Thus the Load Influence due to Static Propoerties of the Software can be shown as:

$$LI_{i\ (static)} \propto \left(\alpha_i + \frac{(N_i + K_i)}{\sum_{i=1}^{tot\ Comp} N_i + \sum_{i=1}^{tot\ Comp} K_i} \right)$$

Or, $$LI_{i\ (static)} = \delta \left(\alpha_i + \frac{(N_i + K_i)}{\sum_{i=1}^{tot\ Comp} N_i + \sum_{i=1}^{tot\ Comp} K_i} \right)$$

where, δ is a constant described ahead in this section.

Usage Pattern Based Parameters

1.F_i – Determines the frequency of use of component C_i in the current run of the software. If a component has been very frequently used in the current execution, then there is a high probability for it to be used in subsequent execution.

(i.e.) $LI_i \propto F_i$

2. U_i – Number of times a component C_i has been loaded in different instances during software startup. If a component has been loaded in more number of instances, then there is a higher probability that it would be used in the next execution too.

(i.e.) $LI_i \propto U_i$

3. T_i – Number of times component C_i was not loaded during software startup. Initial value of T_i is set to be 1. If a component has not been loaded in more number of instances, then there is a lesser probability for it to be loaded the next time too

(i.e.) $LI_i \propto 1/T_i$

Thus, based on the historic load information for a particular component (i.e. U_i and T_i), the Load Influence can be shown to be dependent as:

$LI_i \propto$ Probability of component being loaded historically

(i.e.) $$LI_i \propto \frac{U_i}{U_i+T_i}$$

We associate a constant γ which limits the importance to be given to this factor of a component's startup time load history

Thus, we have:

$$LI_i \propto \gamma \ \frac{U_i}{U_i+T_i}$$

Adding up the importance given due to the frequency of component's current instance usage (limited by constant β), we have:

$$LI_i \propto \left(\beta F_i \ + \ \gamma \frac{U_i}{U_i+T_i}\right)$$

Thus, the Load Influence for a particular component due to dynamism in use of that software can be given as:

$$LI_{i(Dynamic)} \propto \left(\beta F_i \ + \ \gamma \frac{U_i}{U_i+T_i}\right)$$

Or, $$LI_{i(Dynamic)} = (1-\delta)\left(\beta F_i \ + \ \gamma \frac{U_i}{U_i+T_i}\right)$$

Combining the Static and Dynamic Factors of the Load Influence, we get:

$$LI_{i\ (Total)} = LI_{i(Static)} + LI_{i(Dynamic)}$$

(i.e.) $$LI_{i(Total)} = \delta\left(\alpha_i + \frac{(N_i+K_i)}{\sum_{i=1}^{tot\ Comp} N_i + \sum_{i=1}^{tot\ Comp} K_i}\right) + (1-\delta)\left(\beta F_i + \gamma \frac{U_i}{U_i+T_i}\right)$$

Where,

- δ is a constant determining the weight to be given to the 'staticness' of the software, i.e. the design time fixed parameters (i.e. α_i, Ni and Ki).
- Equivalently, (1- δ) determines the weight to be given to the 'dynamism' of the software, i.e. the usage pattern based dynamic parameters (i.e. Fi, Ui and Ti).
- β and γ are the constants denoting importance to be given to the just previous run and to a cumulative history of component's loading.
- A lower β and higher γ implies a higher importance given to historic load data than current data and is useful for finding components in case 1 described earlier.
- A higher β and smaller γ implies a higher importance to be given to the current frequency of use of component C_i. This is useful in the case 2 described earlier.

We compute two LI_i values – one using lower β and higher γ to resemble the case 1 and other using higher β and smaller γ to resemble the case 2.

Values for these constants β, γ, δ (together called Configuration Parameters), would be computed by Simulation which is explained in the next section. Based on the K-Formula described above, we compute Load Influence values for each component for both the cases – History Based Execution as well as Previous Instance Based Execution.

4.3.2 Final Choice of Components to be Included in the Feature-Sets

Once we have both the Load Influence values for each of the components, we compute threshold value for both the cases– Historic Based Execution and Previous Instance Based Execution. Components having higher Load Influence Value than Threshold in

each case are selected to be included in the next application startup based on whether it's a historic based startup or a previous instance based startup.

For Previous Instance Based Case, default threshold is computed by taking mean of all the Load Influence Values computed for this case. And for Historic Based Case, default threshold is computed by taking the half of the mean of all the Load Influence Values for this case.

Apart from this the default computed choice of final components to be loaded can optionally be over ridden by letting the user explicitly mask some particular components that he wants to be loaded or not loaded irrespective of whether or not he uses that component.

Another approach could be to let user define the sensitivity with which to finalize the components. This can be done in form of letting the user move a slider to adjust the sensitivity value which proportionally adjusts the default computed threshold value, thus modifying the threshold, and hence the final count of components that would be loaded. A higher threshold would mean lesser number of components that would be loaded, where as a smaller threshold means a large number of components that will be loaded in the next application startup.

4.4 SIMULATION OF STARTUP UNDER K-METHOD

Simulation is required for two purposes

- To compute the values of the constants used in the K-Formula
- To predict the effective application load time improvement possible by using the K-Method

The values for the constants will be highly specific depending upon following key factors:

- The software itself. (Constants will be different for each software).
- The different usage *scenarios* possible. (discussed below)
- The probability by which the user returns in a short span of time to complete the task being done in previous instance.

For the simulation we need to simulate running several execution instances predicting the components that would be used in the next instance. In each instance we need to randomly request for certain components based on different factors like currently active scenario, wether the user returns to complete previous task or intends to start a new task, and total number of times that the user is requesting for any component. Based on the components selected for loading at software startup, and the actual components requested by the user once the software is run, we calculate the *HIT%* (HIT%: Number of times the requested component is among the loaded components) and *Average components loaded* for each instance, and then for the 500 runs for any configuration (Configuration: a particular combination of β, γ, and δ values). A configuration having higher HIT% and lower number of *Average components loaded*, will be our required optimal configuration for the particular *probJstPrevious Value* (described below). So, a configuration with higher (HIT% - AvgCompLoaded) value will be highly desirable.

4.4.1 Factors Considered for Simulating User's Component Selection Process

To imitate a user's component selection process, we have considered the following factors:

Scenario: A scenario can be considered as a particular usage style of an application leading to a useful and purposeful result.

For Example, A Word Processing Software may involve several scenarios like:

- Usage of spell check, font, colour, workspace components for a usual text editing task,
- Usage of clipart, shapes, image, colour, workspace, page layout, style components for a designer text editing task.
- Usage of workspace, spell check, mail merge component for an email related text editing task

	COMP1	COMP2	COMP3	COMP4	COMP5	COMP6	COMP7	COMP8	COMP9	COMP10
SCENARIO1		✓		✓	✓		✓		✓	
SCENARIO2					✓		✓	✓		✓
SCENARIO3	✓		✓		✓	✓			✓	
SCENARIO4			✓				✓		✓	✓
SCENARIO5			✓		✓	✓	✓		✓	
SCENARIO6		✓		✓	✓		✓	✓		

Table 3: Components in Each Scenario

We have considered six scenarios as described in Table 3. For simulation purpose, we have assumed that the user is free to work on any of the scenarios with equal probability. Thus uniformly distributed random numbers have been used for choosing an active scenario.

ProbJustPrevious*: (Probability of Just Previous Instance)* It defines the probability of the user to re-execute the application to complete the task that he has been doing in the just previous execution of the application. A higher value indicates that the user is continuing the previous task, and a lower value indicates that the user is starting a new task. We have run the simulation for the value of *ProbJustPrevious* ranging from 0 to 1

in steps of 0.1 For each of its values, our aim has been to find the best values of constants β , γ and δ which would give optimal results in terms of getting maximum number of user's requested components from amongst the loaded components. Before the actual deployment of an application employing K-Method Approach, developer is required to set the values of the constants for the K-Formula (discussed above) according to the *ProbJustPrevious* value which best identifies with his software. Alternatively, initial value of *ProbJustPrevious* may be set as 0.5 and a separate module may be included with the software to compute the actual value based on the relative time gaps between consecutive runs of the software over a period of time. Thus, as time would proceed, the *ProbJustPrevious* value would dynamically be computed, and the corresponding configuration (β , γ and δ values) would be used for computation of Load Influence Values through K-Formula.

***#Times Components are to be selected**:* This refers to the number of times that the user will select/use/request components in the particular run of software. Eg: If a user selects component 1 three times, component 5 two times, and component 6 three times, then value for this variable will be 3+2+3=8. This has been simulated through normal distribution of random numbers with mean of 0.5.

probJstPrevious	beta	gama	delta	Hit%	AvgLoadComp	HIT% - AvgCompLoaded
0	0.3	0.8	0.3	0.902527	0.612	0.290527
0.1	0.2	0.9	0.1	0.87574	0.594	0.28174
0.2	0.1	1	0.4	0.850932	0.544	0.306932
0.3	0.1	0.5	0.3	0.845679	0.554	0.291679
0.4	0.2	0.8	0.4	0.860182	0.568	0.292182
0.5	0.2	0.7	0.1	0.844765	0.54	0.304765
0.6	0.1	0.5	0.3	0.864865	0.54	0.324865
0.7	0.8	1	0.4	0.854671	0.53	0.324671
0.8	0.2	1	0.4	0.817544	0.516	0.301544
0.9	0.4	0.8	0.7	0.845938	0.516	0.329938
1	0.3	0.6	0.2	0.838323	0.504	0.334323

Table 4: Final Optimal Configurations and Results Obtained After Simulation

***Getting a Requested Component**:* This refers to a particular component that has been requested by the user in a particular instance of software run. Most of the times this will

be one of the components from the active scenario, but occasionally it can also be some other component not present in the active scenario. For this we have used Normal Distribution considering that 80% of the time component chosen will be from the active scenario (an Interior Component), and 20% of time it could be any other component(an Exterior Component). If a component is being chosen from amongst the active scenario components, then it could be any of those components, so uniformly distributed random numbers have been used for making the final choice of the component from within the active scenario. Alternativly, when not being chosen from the active scenario, component may be chosen from the set of remaining components with uniform probability as well.

In Nutshell, to simulate the above mentioned concept, we have used Monte Carlo Simulation and obtained the results considering the following real world factors:

- Different Scenarios are possible for the user to use.
- If a user returns to application in a short span of time then in most of the cases, he can be expected to be performing his previous task. We have considered that if a user returns in a short span of time, then 80% of the times he would be using the features he used in previous instance; (i.e.) he will be in the same scenario.
- In case he returns to application after a long gap of time, or in the 20% cases of returning in short span of time, a new scenario would be used by him. In that case, he can choose any of the scenarios with equal probability.
- The amount of time gap between consecutive runs of the software can be variable.
- A user may select the same component multiple number of times in the same instance run. Usually a particular set of components is expected to be used more number of times in the same instance (like a text formatting component) and we have used Normal Distribution with mean 0.5 in Simulation to reflect this.
- A user may select a component from among the components that generally define the selected scenario (interior components), or can even choose some other from outside the scenario (exterior components). We have used Normal Distribution with assumption that 80% of the times user will select an interior component and 20% of the times an exterior component will be chosen.

4.4.2 Algorithm for Simulation

***Note**: Values of simulation parameters will be specific from application to application.*

For probJustPrev = 0 to 1 in Steps of 0.1
{
For Beta = 0 to 1 in Steps of 0.1
{
For Gamma = 0 to 1 in Steps of 0.1
{
For Delta = 0 to 1 in Steps of 0.1
{
For k= 0 to 500 //Simulate user's application usage for 500 executions(should be high)
{

- *Determine whether current execution is to be historic based or previous instance based execution*
- *Get the list of components available at startup, and U & T values for all components*
- *Set Active Scenario for Current Execution*
  - *If this is a Previous Instance Based Startup, current Active Scenerio for Application use would be 80% of time the previous Active Scenario, and 20% of the time a new scenario.*
  - *In case of New Scenario, choose the new scenario using Uniform Distribution from the list of existing scenarios and set it as Active Scenario*
- *Simulate User's Requests for Components*
  - *Determine whether user is using an interior component (component already laoded) or an exterior component (component not yet loaded). 80% of times an interior component to the active scenario shall be requested.*
  - *For each component, determine how many times the component would be requested by the user in this instance of application run and update the Frequency Value*
  - *Compute Load Influence Value for Each Component using K-Formula.*
  - *Compute Thresholds and list of Loadable Components for Next Instance Run in the two possiblecases – Historic Based Execution and Previous Instance Based Instance*

}

- *Update the Average HIT% (i.e.) Average Number of Times Component requested was actually already loaded at startup in the 500 instances of execution*

- *Update the AvgCompLoaded Value which determines the average number of components loaded at application startup*

}
}
}
}

4.4.3 Results from Simulation

The results shown in Table 4 can be used in fixing the Configuration Parameter values (beta, gamma and delta) for the known *probJstPrevious* value. On using the obtained values shown in Table 4, optimal choice can be made regarding which components to be loaded for the subsequent execution. Example, from the Table 4, for the *probJstPrevious* (i.e. probability of user returning for completing the task being done in just previous instance) value being 0.6, the value of beta, gamma and delta would be 0.1, 0.5 and 0.3 respectively. On choosing these values, we would need to load just 54% of the components, thus reducing load time by 46% (assuming that all components take equal time to load), HIT% would be 86.49 (i.e. 86.49% of the times, a component requested by the user shall be already available amongst the already predicted and loaded components). The remaining 17.51% of the components will need to be dynamically loaded and should be an acceptable trade-off against the benefit obtained from almost 50% reduction in load time.

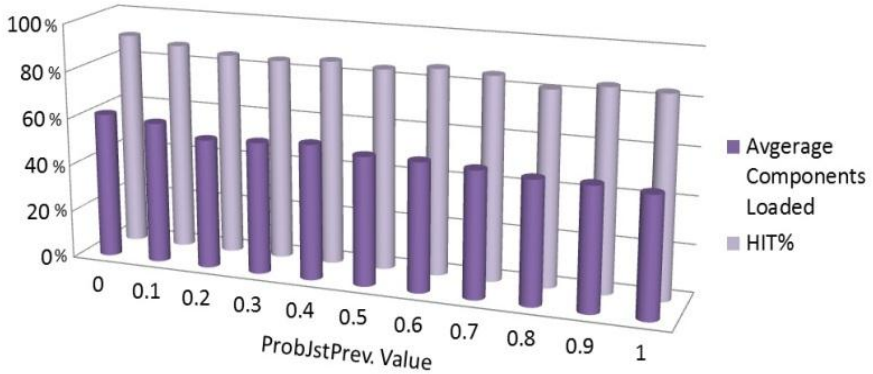


Figure 11: Graph showing variation of Average Numbers of Components Loaded, HIT% against ProbJstPrevious

V. TEST APPLICATION BASED ON K-METHOD – 'ADVANCE BROWSER'

To prove the efficiency of the K-Method in an actual application, we have developed a Browser Application and named it 'Advance Browser'. Our aim has been to have an application with basic navigation functionalities and to have some external components providing additional functionalities which can be attached to the main browser application in form of plugins. During development time, we needed to run a simulation to determine the configuration parameters (beta, gamma, delta) to be used in the K-Formula for choosing the components to be loaded at startup.

For better understanding of K-Method in execution, we have provided an 'Illustration Window' which displays the Load Influence Values being computed for each component as the user interacts with the application. It also displays the startup time for that particular instance as well as the average startup time since the application install. Further, it displays a dynamic list of components that shall be loaded in the next startup; the list would change as the user uses the application based on which components he is currently using and how much he uses them.

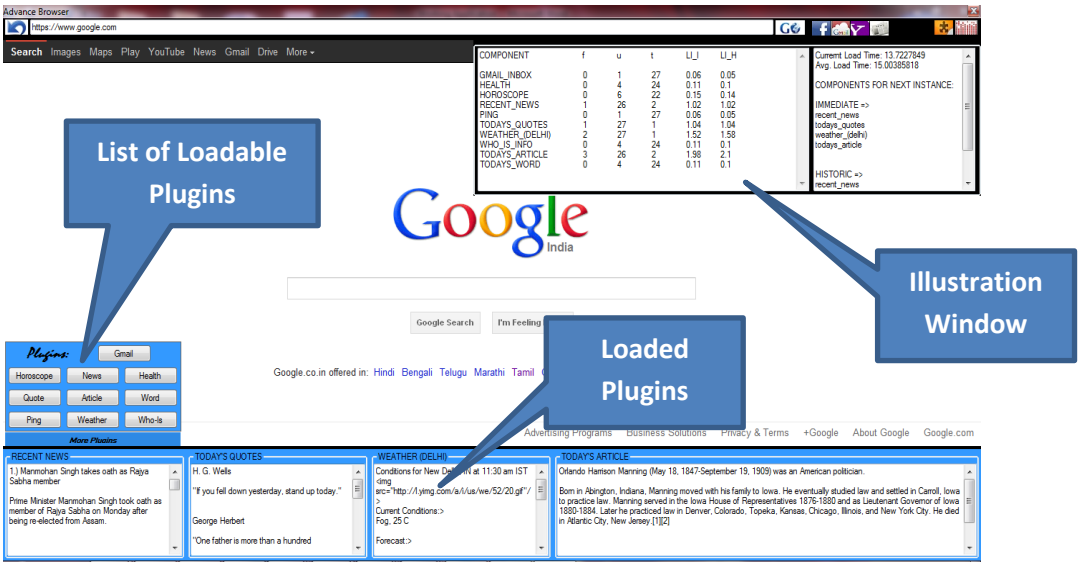


Figure 12: Snapshot of the Test Application - 'Advance Browser'

5.1 APPLICATION ARCHITECTURE

Our Test Application, developed in VB .NET 2010, consists of a basic browser module which acts like a host module and 10 plugins which add up to functionality of the browser by providing different features.

A particular set of plugins is loaded at application startup and in case the user wishes to use some feature provided by a plugin not yet loaded, then it can be dynamically loaded.

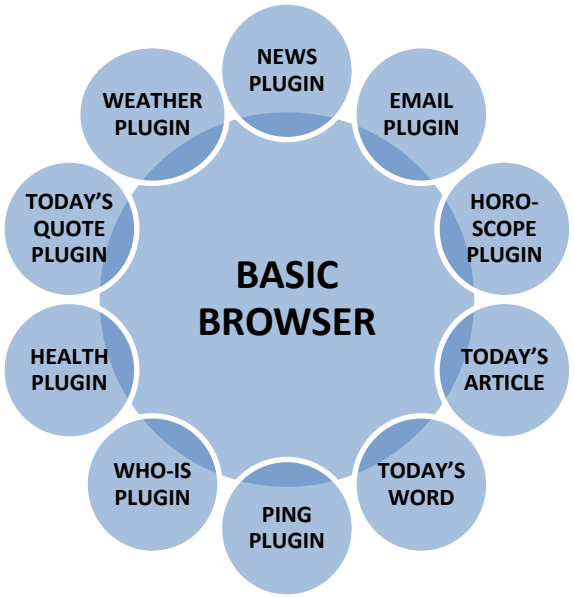


Figure 13: Plugins Associated with the Basic Application

The 10 plugins developed for our application are described below:

Recent News Plugin: This fetches the top news stories from the internet .

Email Plugin: This fetches the unread emails from a previously registered mail account. If there is no unread mails, then it fetches the 5 most recent mails.

Horoscope Plugin: This fetches the current day's horoscope for all the zodiac signs

Today's Article Plugin: This plugin fires request to Wikipedia and retrieves a random article from it.

Today's Word Plugin: This plugin displays a random word of the day and its meaning

Today's Quote Plugin: This plugin displays a random quote of the day

Ping plugin: This can be used to issue ping commands directly. It is set as default to ping our college homepage http://www.dce.edu/

WhoIs plugin: This can be directly used to issue WhoIs command for a website. It is set as default to enquire about https://www.google.com

Health Plugin: This fetches the top health related bulleton and displays to the user

Weather Plugin: This displays the current day weather conditions for a city. By default it is set to show weather conditions of New Delhi, India.

These plugins exists as DLL files and can be dynamically and selectively loaded either at application startup or during program run time.

Baisc Features of our Host Application are:

- Basic browsing functionality including an Address Bar and Bookmarks
- Illustrator Window: To illustrate functioning of the K-Formula computations and to show the choices being made for loading components during next startup
- Plugin Display Pannel: To display the results from the operation of different plugins
- Buttons to Hide the Illustrator Window and the Plugin Display Pannel for performing normal browsing task

An important factor in our application is that the user's application usage pattern is continuously monitored. For each interaction of user with any component, the frequency of use for that component is increased accordingly. Load Influences and Threshold Values are re-computed, and the list of components to be loaded in subsequent startup is updated as well.

An XML File is used to keep track of the Alpha, N, K, U, T and Frequency values for each component. The same file stores Boolean values for whether component needs to be loaded or not in next instance for the two cases – Historic Based and Previous Instance Based.

5.2 WORKING OF APPLICATION DURING DIFFERENT PHASES UNDER K-METHOD

Activities of our Browser Application for different phases under K-Method can be understood from the below flowchart:

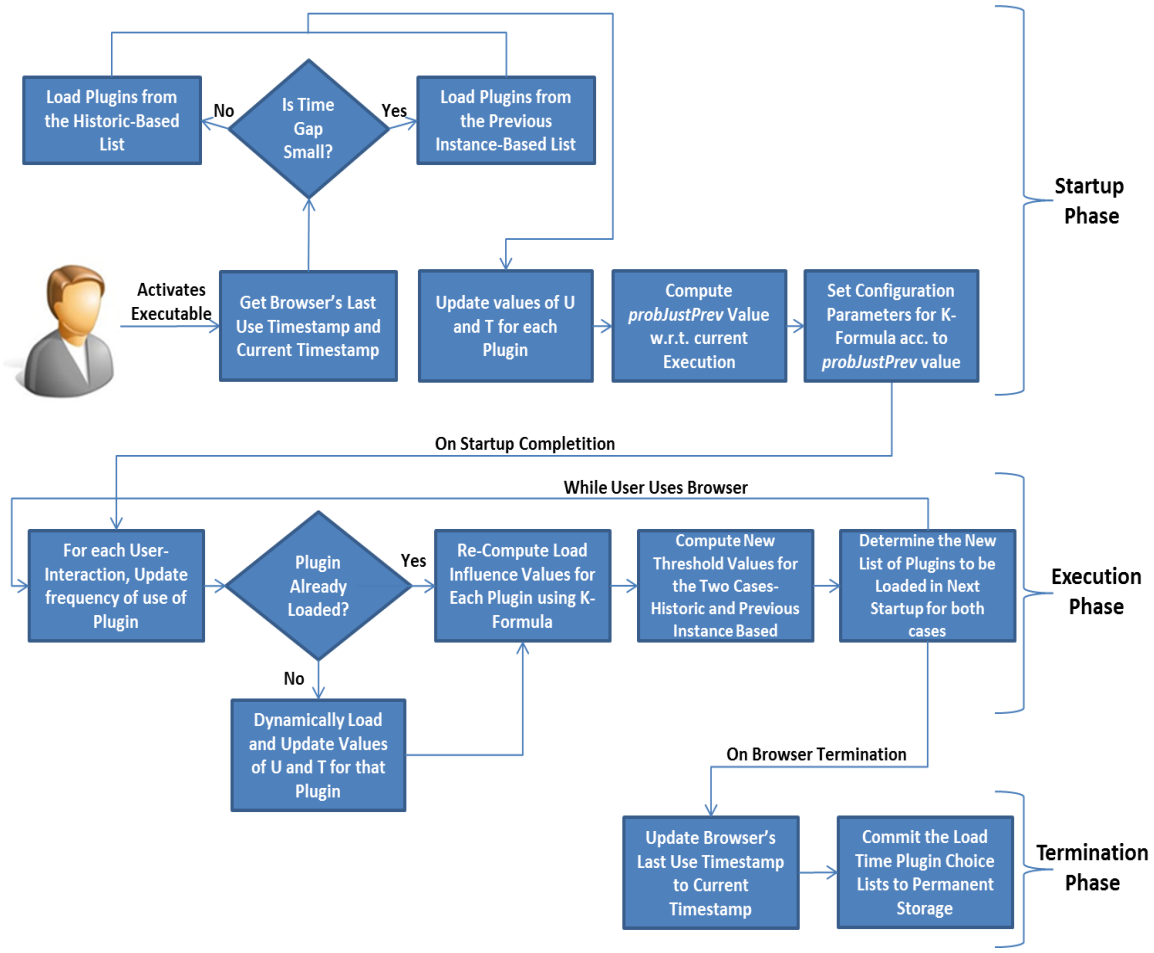


Figure 14: Flowchart of Our Application Working under K-Method

Browser Application Startup Phase: When the user clicks on the executable, the browser load time initialization routine is called which determines the current timestamp, and retrieves the timestamp of the browser's previous use from the configuration settings. It computes the time gap and in case it is less than 2 hours (modifiable value from developer's end), then the components specified in the Previous Instance Based Component List are loaded. On the other hand, if the time gap was larger, then the components specified in the Historic Based Component List are loaded. For all the components (loaded or not loaded), the value of U and T are updated. As discussed in

earlier section, U defines the number of instances in which a component was loaded since application install, and T defines the number of instances in which a component was not loaded since application install. Thereafter, value for *probJustPrev* is computed which is a ratio that shows that since the time application has been installed how much proportion of times has the application been loaded in a short span of time from its previous termination leading to application being started in a previous instance based case. Based on the value obtained for *probJustPrev*, the configuration parameters of K-Formula are given corresponding values obtained from simulation during development time.

Browser Application Execution Phase: Once the browser is loaded, the user continues to use the browser and associated plugins as per his requirements. His interactions with the features provided by the plugins are monitored continuously and each time he uses a particular plugin, the value of frequency of use for that particular plugin is incremented. It should be noted that initially, at application startup, value of frequency of use for each plugin is set to zero and this continues to increase as the user uses the application. In case the user wishes to use a feature provided by a plugin not yet loaded, then it would be dynamically loaded and the U and T values for it will be updated accordingly.

Browser Application Termination Phase: When the user closes the browser, then two main things happen. Firstly, the current timestamp value is noted and saved in the field for browser's previous use timestamp value in the configuration file. Secondly, the list of components to be loaded at next startup, for the two cases (Historic Based and Previous Instance Based), is committed to the permanent storage in the XML File.

5.3 SIMULATION TO COMPUTE CONFIGURATION PARAMETERS FOR K-FORMULA

For using the K-Formula, we need to first find out the Configuration Parameters (beta, gamma, delta) which would be used with it. Values of these parameters would be different for different values of *probJustPrev*. Using Simulation, we would find out the values that give us the most optimal results. i.e. simulation of application usage scenarios under optimal parameters should result in a case where most of the user requests for features are from the set of components offering those features and already loaded during startup.

Factors Considered for Simulation of our Browser Application

Scenarios: We have considered 6 different scenarios for usage of our test application. These are described as below:

COMPONENTS	Email	Horoscope	News	Health	Quote	Article	Word	Ping	Weather	Whois
Scenario 1	✓		✓					✓		✓
Scenario 2		✓			✓	✓	✓			
Scenario 3	✓	✓	✓						✓	
Scenario 4			✓		✓	✓	✓			
Scenario 5	✓		✓	✓	✓				✓	
Scenario 6	✓	✓	✓		✓					

Table 5: Components Used in Different Scenarios for 'Advance Browser'

Scenario 1: This describes expected usual usage by a Computer Savvy User. We have assumed that among the available plugins, an average computer programmer would most of the times would be using features of Gmail, news, ping and whois.

Scenario 2: This describes expected usual usage by a Literature Loving User. We have considered features of quote of the day, word of the day, article of the day and horoscope for such a user.

Scenario 3: This describes expected usual usage by some user in case of an event like a Natural Calamity in town. A user in this case might be expected to use his Email to

communicate and at the same time he can be expected to check News and Weather Reports very frequently and also occasionally check his horoscope.

Scenario 4: This describes expected usual usage by a user preparing for delivering some speech or debate. We have included features of news, quote, word and article of the day in this scenario.

Scenario 5: This describes expected usual usage by a doctor. We have included features of Health, Email, News, Quote of the Day, and Weather in this scenario.

Scenario 6: This describes executed usual usage by an aspiring politician. He would check his mails and news regularly. He would also be interested in quotes that he can use in his speech. At same time he can be expected to check his horoscope on regular basis.

NOTE: The scenarios mentioned above should ideally be found after a decent survey involving recording of features that the users usually use and what kind of tasks they usually do in similar applications while using those features.

On competition of Simulation we have obtained the following optimal parameters for *probJustPrev* values ranging between 0 and 1 at steps of 0.1:

ProbJstPrev	BETA	GAMMA	DELTA	HIT%	AvgCompLoaded	HIT%-AvgCompLoaded
0	0.1	0.7	0.3	0.88	0.54	0.34
0.1	0.1	0.5	0.2	0.87	0.54	0.33
0.2	0.1	1	0.2	0.87	0.55	0.32
0.3	0.1	1	0.4	0.86	0.50	0.35
0.4	0.2	0.8	0.1	0.85	0.50	0.35
0.5	0.4	1	0.1	0.88	0.52	0.36
0.6	0.1	0.6	0.2	0.88	0.50	0.37
0.7	0.2	0.8	0.1	0.85	0.48	0.37
0.8	0.8	0.9	0.4	0.86	0.52	0.34
0.9	0.5	1	0.2	0.86	0.49	0.37
1	0.6	0.4	0.8	0.85	0.45	0.40

Table 6: Final Optimal Configuration Values Obtained After Simulation

The simulation also suggests that on an average, 51% of the components shall be loaded and 86% of the times user would use a component already loaded during startup.

5.4 THE EXPERIMENT

To verify the simulation results and hence correctness of our methodology, we have performed a simple experiment on our test application.

In First Case we have executed our application under K-Method 20 times and noted the StartUp time, and the Memory Consumed.

In Second Case we have executed our application under "Full Execution Mode" which loads all the components at application startup. This is equivalent to our application running without K-Method.

We have then computed the average startup time and average memory requirements for our application running under K-Method. We have also computed the average startup time and average memory requirements for our application when run without K-Method.

For computing the startup time, we have placed timestamps within the application code at point where the application first loads and at point where the startup time specified components have been loaded and initialized.

For knowing the memory requirements, we have checked the memory consumed by our process in Windows Task Manager

The above experiments have been performed on a Dell Laptop running 64 bit Windows 7 Home Premium OS, and having I5 2.40GHz Processor, 4GB RAM and 500GB Hard Disk Drive

5.4.1 CASE 1: Application Running Under K-Method

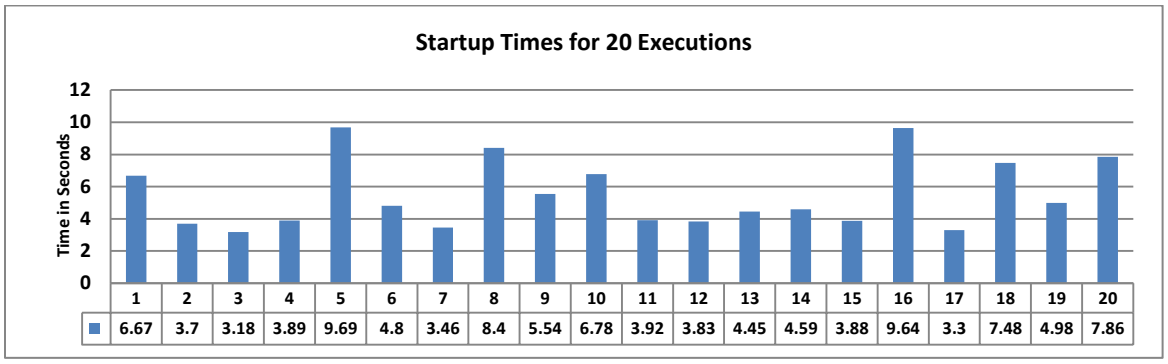


Figure 15: Startup Time for Test Application (under K-Method) in 20 Instances

As shown in Figure 15, startup time has varied from a minimum of 3.3 seconds to a maximum of 9.69 seconds. The graph is uneven because the startup time would depend upon the number of components being loaded at startup which would vary depending upon the user's usage pattern. Moreover different components would require different times for their initializations, so startup time would not only depend upon the number of components being loaded, but also on the time required for initialization code by a particular component being loaded. Thus, it is possible that one larger component consumes more time than three smaller components put together. Average Startup time is 5.50 seconds.

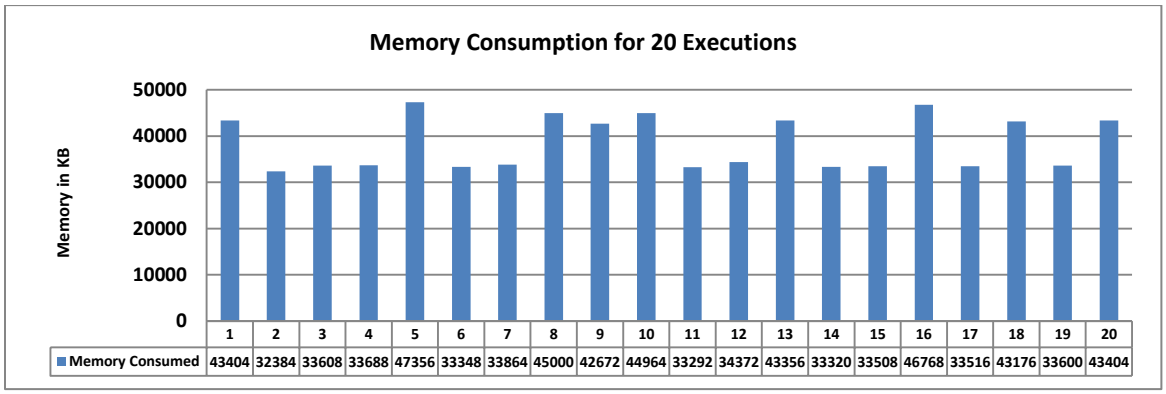


Figure 16: Memory Req. at Startup of Test Application (under K-Method) in 20 Instances

Similarly, in Figure 16, we can see that memory is varying from a minimum of 3329 KB to a maximum of 4735 KB. The variation of memory consumption can be accounted to the variation in memory requirements of individual components being loaded. Average Memory Requirement is 38430 KB.

5.4.2 CASE 2: Application Running without K-Method (i.e. Conventional Manner)

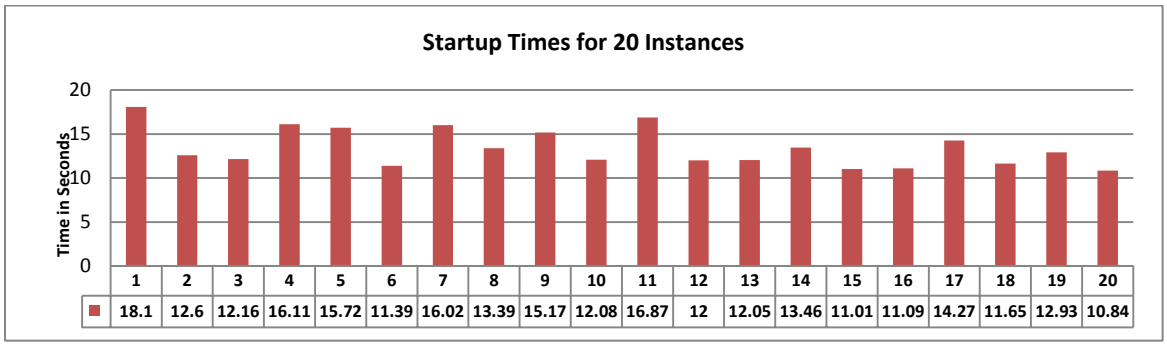


Figure 17: Startup Of Test Application (Without K-Method) for 20 Instances

From Figure 17 we see that minimum startup time is 10.84 seconds and maximum startup time is 18.1 seconds and average time is 13.44 seconds. Since all the components are being loaded in all the instances yet there is variation in startup time. This, we assume, is occurring due to the caching being done by the OS and browser in the background. Some components used in our application make use of network connection during the initialization process. This can lead to different time requirements in different instances due to presence or absence of data in the OS level or the browser level cache.

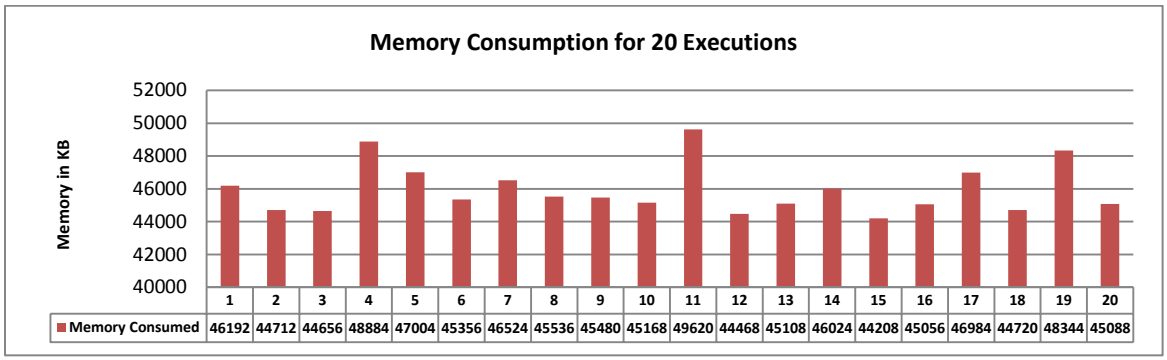
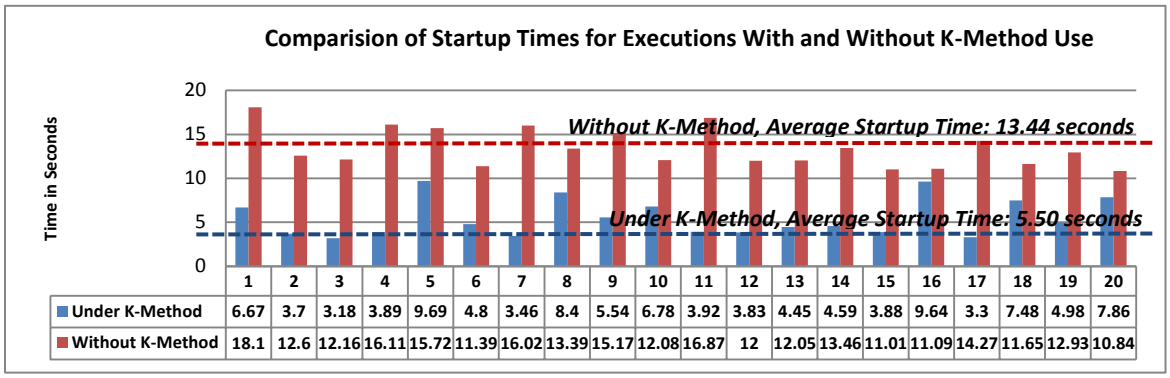


Figure 18: Memory Req. at Startup of Test Application (without K-Method) in 20 Instances

From Figure 18 it can be seen that minimum memory consumed is 44208 KB and maximum memory consumed is 49620 KB at average of 45956.6KB. Ideally no variation should have been there since all the components are being loaded in all the instances. However, variation can be attributed to the fact that the memory under consideration is the Private Working Set Memory. If a component is currently holding some sharable memory, then only it would be reflected in the private working set of our application and not if some other process is holding that part of memory.

5.4.3 Analytical Comparison of Results for Case 1 and Case 2

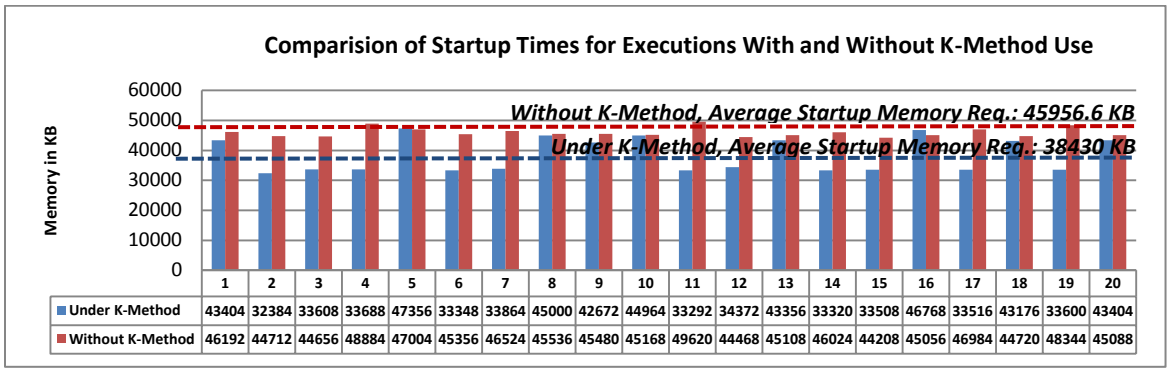
Below we have compared the results of startup times and memory consumption for test application when running under K-Method and when running without K-Method.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Under K-Method	6.67	3.7	3.18	3.89	9.69	4.8	3.46	8.4	5.54	6.78	3.92	3.83	4.45	4.59	3.88	9.64	3.3	7.48	4.98	7.86
Without K-Method	18.1	12.6	12.16	16.11	15.72	11.39	16.02	13.39	15.17	12.08	16.87	12	12.05	13.46	11.01	11.09	14.27	11.65	12.93	10.84

Figure 19: Comparison of Startup Times for Test Application With & Without K-Method

As evident from the graph, on using K-Method, startup time has improved by about 59.07% by dropping in average from 13.44 seconds to 5.50 seconds. This is in agreement with the results found in simulation, where it was found that on using the suggested configuration parameter values, about 51% of the components will need to be loaded and approximately 86% of the times, user would request from the already loaded components.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Under K-Method	43404	32384	33608	33688	47356	33348	33864	45000	42672	44964	33292	34372	43356	33320	33508	46768	33516	43176	33600	43404
Without K-Method	46192	44712	44656	48884	47004	45356	46524	45536	45480	45168	49620	44468	45108	46024	44208	45056	46984	44720	48344	45088

Figure 20: Comparison of Memory Req. for Test Application With & Without K-Method

We can see that K-Method not only assists in improving startup time of an application but it also assists in memory consumption of an application during application startup. On using K-Method an improvement of 16.37 % in memory requirement was obtained with memory consumption dropping to average of 38430 KB from 45956.6 KB when K-Method was not used. This can be attributed to not loading the components that user is not expected to use.

VI. CONCLUSION

Improving startup times of applications is a problem that has interested researchers since a long time. Faster processor and faster secondary storage media has led to an overall improvement in user experience over the decades. From the end of the software developers also efforts have been made to optimize the codes to take lesser resources of space and time. However, not much had yet been done with respect to observing user's application usage pattern and then determining the components that user is expected to use in the next execution. A very simple intuitive idea suggests that if we load lesser components during startup, then lesser time would be taken for application to startup and also smaller memory would be consumed by application. However, making this correct prediction of the components expected to be used in next execution is a very important as well as a limiting factor. We have given a concept called K-Method using which makes use of K-Formula to make this decision of whether or not to load a particular component at startup. We have performed simulation and results suggested us a possibility of improving load time by at least 46%. We then developed a test application called 'Advance Browser' which is a browser with basic navigation functionality and other advanced functionalities provided by separate plugins. Our experiments with our test application showed about 59% improvement in startup time and about 16% improvement in memory consumed at application startup.

Based on the results, it is highly recommended that large softwares should be developed while making use of this approach for reducing the startup time. Use of this approach is a onetime investment of time. This one time investment by the developers will lead to tremendous reduction in software load time which will be highly beneficial for any potential user and hence also will add creditability to the development team for having developed 'faster' software.

VII. FUTURE WORK

Our proposed method can also be used in other domains where the system is composed of components and the components can be selectively loaded or initiated. One such domain is embedded systems where resources like memory and battery life are very constrained and limited. K-Method can be used to predict which components user is expected to utilize in next execution. Software components would lead to saving up on memory requirements and hardware components would lead to saving up of battery life when these are selectively powered on. This can potentially be a very efficient technique to save battery life in this smartphone era.

When applied to selectively load components that communicate over the network, our method can also result in saving considerable bandwidth. This is especially true in case of browsers where user installs several add-ons but seldom uses them all. During browser startup, these add-ons not only consume time and memory, but also considerable bandwidth which our method can considerable reduce by not loading the components that are not expected to be used.

VIII. REFERENCES

[1] Oliver W. Steele, David T. Temkin, P. Tucker Withington Adam G. Wolff, "System for optimizing application start-up. ," 10/720,726, 2009.

[2] Andrew W. Wilson and W. David Schwaderer, "Understanding I/O subsystem," *Adaptec Press*, pp. 47-80, 1996.

[3] Flanagan J. K Yin N., "Reducing application load time by rearranging disk data," 1998.

[4] Paul Vongsathorn and Scott D. Carson, "A System for Adaptive Disk Rearrangement," *Software---Practice and Experience*, 1990.

[5] Sedat Akyuerek and Kenneth Salem, "Adaptive Block Rearrangement Under UNIX," *Software---Practice and Experience*, 1997.

[6] Xiao-Hong Tu and Niki C. Thornock and J.Kelly Flanagan, "A Stochastic Disk I/O Simulation Technique," in *Proceedings of the 1997 Winter Simulation Conference*, 1997, pp. 1079-1086.

[7] Benjamin Aaron Rudiak-Gould James Edward Walsh, "System and Method for Improved Program Launch Time," US 6202121 B1, 2001.

[8] Youngjin Cho, Kyungsoo Lee, and Naehyuck Chang Yongsoo Joo, "Improving Application Launch Times with Hybrid Disks," in *CODES+ISSS'09*, 2009.

[9] (2006) [Online]. http://www.microsoft.com/whdc/system/sysperf/

[10] Junhee Ryu, Sangsoo Park, and Kang G. Shin Yongsoo Joo, "FAST: Quick Application Launch on Solid-State Drives," in *9th USENIX Conference on File and Storage Technologies.*, 2011.

[11] Tim R. Bird, "Methods to Improve Bootup Time in Linux," in *Linux Symposium*, 2004, pp. 79-88.

[12] Tim, and Frank Yellin. Java virtual machine specification. Addison-Wesley Longman Publishing Co., Inc., 1999. Lindholm,.: Lindholm, Tim, and Frank Yellin. Java virtual machine specification. Addison-Wesley Longman Publishing Co., Inc, 1999.

[13] Lucent Technologies. Inferno. [Online]. http://inferno.bell-labs.com/inferno/

[14] A., Langdale, G., Lucco, S. and Wahbe, R Adl-Tabatabai, "Efficient and Language Independent Mobile Programs.," in *Conference on Programming Language Design and*, 1996.

[15] Brockschmidt, "Inside OLE. ," 1994.

[16] A., Sethi, R. and Ullman, J. Aho,. New York: Addison-Wesley, 1986.

[17] E. and Graham, S. L. Pelegri-Llopart, "Code Generation for Expression Trees: AnApplication of BURS Theory.," in *15th ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, San Diego, CA, 1988, pp. 294-308.

[18] C., Calder, B., Lee, H. P. and Zorn B. G. Krintz, "Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs," in *Architectural Support for Programming Languages and Operating Systems*, 1998.

[19] D., Baer, J. L., Bershad, B. N. and Anderson, T. Lee, "Reducing Startup Latency in Web and Desktop Applications," , 1999.

[20] E., Arthur J. Gregory, and Brian N. Bershad Sirer, "A practical approach for improving startup latency in Java applications. ," , 1999.

[21] Pramote, Vincent J. Mooney, and Vijay K. Madisetti Kuacharoen, "Software Streaming via Block Streaming," in *Design, Automation and Test* , 2003.

[22] Jelinek, Jakub, "Prelink," 2004. [Online].

[23] John Richard Moser. (2006) [Online]. http://lwn.net/Articles/192624/

[24] Ulrich Drepper, "How to write shared libraries.," 2006.

[25] Michael Meeks. (2006) Speeding up the dynamic linker.

[26] Heeseung, Hwanju Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng Jo, "Optimizing the startup time of embedded systems: A case study of digital TV," in *Consumer Electronics, IEEE Transactions on 54, no. 4*, 2009, pp. 2242-2247.

[27] Hiroki Kaminaga, "Improving Linux Startup Time Using Software Resume (and other techniques)," *Linux Symposium*, p. 17, 2006.

[28] Inwhee Joe and Sang Cheol Lee, "Bootup time improvement for embedded Linux using snapshot images created on boot time," in *Next Generation Information Technology (ICNIT), The 2nd International Conference on*, 2011, pp. 193-196.

[29] B. Esfahbod, "Preload—An adaptive prefetching daemon ," University of Toronto, Doctoral dissertation 2006.

[30] Changwoo Min, Jeehong Kim, Young Ik Eom Hokwon Song, "Usage Pattern-Based Prefetching: Quick Application Launch on Mobile Devices," in *Computational Science and Its Applications – ICCSA*, vol. 7335, 2012, pp. 227-237.

[31] Gary V. Vaughan. (2006) Autoconf, Automake, and Libtool.

[Online]. http://sourceware.org/autobook/autobook/autobook_158.html

[32] J.H.M.Dassen. (1995) LinuxU.

[Online]. http://linux4u.jinr.ru/usoft/WWW/www_debian.org/Documentation/elf/node7.html

[33] IBM. (2012) Program execution model.

[Online].http://pic.dhe.ibm.com/infocenter/aix/v7r1/index.jsp?topic=%2Fcom.ibm.aix.prftungd%2Fdoc%2Fprftungd%2Fperf_overview.htm

[34] Joanna McGrenere, "“Bloat”: The Objective and Subject Dimensions," in *CHI'00 extended abstracts on Human factors in computing systems*, 2000, pp. 337-338.

[35] Vince Baskerville. (2011) Build a product. Not Features.

[Online]. http://vincentjordan.com/2011/06/build-a-product-not-features/

[36] Microsoft Corporation. (2012) Installing Visual Studio 2012.

[Online]. http://msdn.microsoft.com/en-us/library/vstudio/e2h7fzkw.aspx

[37] CodeBlocks, *CodeBlocks Manual*., 2010.

[38] Mozilla. (2010) Mozilla Plugin Support on Microsoft Windows.

[Online]. http://plugindoc.mozdev.org/

[39] Matt Smith. (2011) [Online]. http://www.digitaltrends.com/computing/a-beginners-guide-to-google-chrome-why-its-time-to-ditch-internet-explorer/

[40] Microsoft Corporation. (2007) Enable or disable add-ins in Office programs.

[Online].http://office.microsoft.com/en-in/help/enable-or-disable-add-ins-in-office-programs-HA010034127.aspx

[41] John Wayne Hill. (2010) Perceived Speed Performace.

[Online]. http://www.johnwaynehill.com/blog/2010/06/16/perceived-speed-performace/

[42] Amit Agarwal. (2008) Installing Software? Know What Happens Behind The Scene.

[Online].http://www.labnol.org/software/tutorials/fix-install-errors-troubleshoot-software-installation-problems/2841/

[43] codecoffee. Understanding software Installation.

[Online]. http://www.codecoffee.com/tipsforlinux/articles/27.html

[44] MozillaSupport. (2012) Find and install add-ons to add features to Firefox.

[Online]. http://support.mozilla.org/en-US/kb/find-and-install-add-ons-add-features-to-firefox

[45] Paul Placido Giangarra, Ravindranath Kasinath Manikundalam, Donald Robert Padgett, James Michael Phelan James Wendell Arendt, "System and method for lazy loading of shared libraries," US 5708811 A.

[46] StackOverflow. (2009) What happens when you run a program?

[Online]. http://stackoverflow.com/questions/1204078/what-happens-when-you-run-a-program

[47] NovelSupport. (2002) What happens when an application is launched?

[Online]. http://support.novell.com/docs/Tids/Solutions/10022703.html

[48] Michael Muchmore. (2013) Which is the Fastest Browser.

[Online]. http://www.itproportal.com/2013/01/04/which-is-the-fastest-browser/

[49] Whitson Gordon. (2013) Browser Speed Tests. [Online]. http://lifehacker.com/5976082/browser-speed-tests-chrome-24-firefox-18-internet-explorer-10-and-opera-1212

[50] newrelicblog. (2012) Which Browsers are the Fastest? [Real User Performance Data].

[Online]. http://blog.newrelic.com/2012/04/05/fastest-browsers/

[51] MozillaZine. (2007) Increasing startup speed.

[52] Mozilla. (2011) All about Performance. [Online]. http://blog.mozilla.org/tglek/2011/05/13/firefox-telemetry/