The Top Ten List: Dynamic Fault Prediction

Ahmed E. Hassan and Richard C. Holt
Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo
Waterloo, Canada
{aeehassa,holt}@plg.uwaterloo.ca

ABSTRACT

To remain competitive in the fast paced world of software development, managers must optimize the usage of their limited resources to deliver quality products on time and within budget. In this paper, we present an approach (*The Top Ten List*) which highlights to managers the ten most susceptible subsystems (directories) to have a fault. Managers can focus testing resources to the subsystems suggested by the list. The list is updated dynamically as the development of the system progresses.

We present heuristics to create the Top Ten List and develop techniques to measure the performance of these heuristics. To validate our work, we apply our presented approach to six large open source projects (three operating systems: *NetBSD*, *FreeBSD*, *OpenBSD*; a window manager: *KDE*; an office productivity suite: *KOffice*; and a database management system: *Postgres*). Furthermore, we examine the benefits of increasing the size of the Top Ten list and study its performance.

1 INTRODUCTION

Managers of large software projects need to ensure that their projects are delivered within budget with minimal schedule slippage. They have to prevent the introduction of faults, ensure their quick discovery and immediate repair, and make sure the software can evolve gracefully to handle new requirements from customers. Unfortunately, these demands are fulfilled with restricted personnel resources within limited time. Resource allocation becomes a non-trivial challenge which managers must face.

Managers would like to optimize the usage of their resources. They would like to allocate resources to areas that are in need of these resources and reassign them as soon as interest and focus shifts. In this paper, we address the challenges surrounding fault detection and repair in large software systems. We would like to give managers a *Top Ten List* of subsystems that are most susceptible to faults. We need the list to be updated dynamically to reflect future risks based on the current status of the project. By limiting the number of subsystems in the list, we hope to give managers an easy and clear way to allocate their limited resources. By updating the list as the software system evolves and the risks associated with components change, we hope to give managers a dynamic tool which can consistently give informed and up-to-date warnings. Finally, we would like to build a tool that is not intrusive and requires as little details and setup as possible to permit managers to get a high return on their investment.

Previous research in software faults has focused on two areas:

1. **Count** based techniques which focus on predicting the number of faults in subsystems of a software system. Managers can use these predictions to determine if the software is ready for release or it has many lurking bugs. They can use the predictions to guide their resource allocations as they wind up the project towards release. These models are validated by dividing the data into equal size periods and predicting the faults in one period using data from all the previous period. For example, the fault data from one release can be used to predict faults in following releases. Examples of such work are Graves *et al.* [6], Ohlsson and Alberg [11], and Schneidewind [14].

2. **Classification** based techniques which focus on predicting which subsystems in a software system are fault-prone. Fault-prone is defined by a manager, for example a fault prone subsystem may be any subsystem with more than two faults in a release. These predictions can be used to assist managers in focusing their resources allocation in a release, by allocating more testing resources and attention to fault-prone subsystems. Again these models are validated by testing if the data from one release can be used to predict if a subsystem is fault prone in following releases. An example of such work is Munson and Khoshgoftaar [10].

These approaches focus on long term planning. They are designed for long term prediction and are validated by building statistical prediction models which use data from one software release to predict values in following releases. In this paper we focus on short term dynamic prediction. We present an approach to validate these short term predictions and we show an analysis of this approach using several heuristics for fault predictions. This focus on short term planning would permit managers to monitor more closely the development and testing processes instead of simply depending on long term planning which tends not to react well to varying competitive market pressures over the lifetime of a software system.

We focus on predicting the subsystem that are most likely to contain a fault in the near future, in contrast to count based techniques which focus on predicting an absolute count of faults in a system over time, or classification based techniques which focus on predicting if a subsystem is fault prone or not. For example, even though a subsystem may not be fault prone and may only have a few number of predicted faults, it may be the case that a fault will be discovered in it during the next few days or weeks. Or in another case, even though a fault counting based technique may predict that a subsystem has a large number of faults, they may be dormant faults that are not likely to cause concerns in the near future. If we were to draw an analogy to our work and rain prediction, our prediction model focuses on predicting the areas that are most likely to rain in the next few days. The predicted rain areas may be areas that are known to be dry areas (*i.e.* not fault prone) and may be areas which aren't known to have large precipitation values (*i.e.* low predicted faults).

The predictions are presented to managers as a list of the Top Ten most likely subsystems to have faults. That list is modified over time as new files are modified within a subsystem or as new faults are discovered and fixed. To validate the quality of our predictions, we borrow concepts from the vast literature of caching – file system and web proxy caching. In particular, we use the idea of *Hit Rate* traditionally used to determine the quality of caching systems. A high Hit Rate indicates that the Top Ten list is performing well and subsystems with recently discovered faults had been already present in the list. Moreover, we present new metrics – *Average Prediction Age* – to measure the practical benefits of predictions in the Top Ten list. A prediction that warns of a fault occurring within a couple of hours is not as valuable as a prediction that warns of a fault a couple of weeks before its occurrence. A prediction that warns of a fault occurring within a couple of years and misses predicting more recent faults is not valuable either.

Organization of Paper

The paper is organized as follows. Section 2 introduces the motivation behind our work and explains the concepts of *Hit Rate* and *Average Prediction Age*. We use both concepts to evaluate and compare different fault prediction heuristics presented in this paper. In Section 3, we present several heuristics to build the Top Ten List based on various characteristics. Then in Section 4, we present short introductions to each of the six open source systems used in our case study. In Section 4, we measure the performance of the proposed heuristics by analyzing the development history of the studied software systems using the *Hit Rate* and *Average Prediction Age* concepts introduced earlier. Later in Section 6 we analyze the performance benefits of increasing the size of the proposed Top Ten list. In Section 7, we discuss our results and address shortcomings and challenges we uncovered in our approach. Section 8 showcases related work in the fault prediction literature. Finally, section 9 summarizes our results and presents plans for future work.

2 MOTIVATION

To cope with a large number of tasks at hand, managers are always in search of a silver bullet that would give them a list of issues to focus their limited resources on. Hence, the idea of the Top Ten list. The Top Ten list is a list of the top ten subsystems which are most susceptible to have a fault appear in them in the near future. Managers can use this list to focus their limited resources and maximize their resource usage.

The inspiration of the idea of Top Ten list comes from the idea of a resource cache. Previously, caching has been proposed to solve many problems associated with limited resources and latency associated with acquiring them. In the file system domain, caching is used to store previously used files in memory so future requests to these files would be fulfilled from memory instead of accessing the hard drive which is much slower than memory. The same ideas and concepts have been applied to database and web systems.

Conceptually, a cache is used to store a limited number of resources for cheap access. Heuristics employed by the cache system determine which resources to store, usually based on the probability that the resource will be accessed in the near future. For example, in a file system cache it is expected that a file that was accessed recently will be accessed again within the next few minutes. By storing this file in the cache, consecutive accesses will be much faster as they won't require slow disk access. Unfortunately, a cache is usually a limited resource. For example memory is much smaller than a hard drive, or a web proxy server is much smaller than the whole Internet. Thus cache replacement heuristics are used to decide which resources should stay in the cache and which ones should be evicted to store new cacheable resources.

We believe the same idea can be adopted for deciding which subsystems are most susceptible to having a fault in the near future. A manager of a project can only focus on a limited number of resources. These limited resources can be thought off as the cache system size. Cache heuristics can be developed to determine which subsystems are no longer susceptible to a fault and which are still susceptible to a fault. For example, research has shown that previous faults in a subsystem are good indicator of future faults [5]. One heuristic would build the Top Ten list based on the number of previously discovered faults in a subsystem. Thus the Top Ten list would contain the ten most faulty subsystems. Other heuristics based on the number of developers that worked on the subsystem, the recency of the latest fault or modification, the size of the subsystem, the number of modifications, or a metric which combines some of these ideas are a few of the possible heuristics.

The rich literature of fault analysis and prediction can be used to develop heuristics and many of previous fault prediction findings can be validated using our presented approach. Fenton and Neil organize defect prediction models based on the source of the data used for the prediction into three main areas [5]:

1. Models based on size and complexity metrics
2. Models based on testing metrics
3. Models based on process quality metrics.

In addition, work by Graves *et al.*[6] and Munson and Khoshgoftaar [7] suggest using code change metrics such as code churn [3] to build quality prediction models. Any of these aforementioned model data can be used to build the Top Ten list. In particular, models based on size and complexity, or models based on the code change process are the most promising ones to build the Top Ten list, as the value of their metrics tends to change over the short term as source code is modified to enhance the software system. In contrast models based on process quality metrics such as CMM ratings tend to be more stable and would be more useful for long term predictions.

By basing the idea of Top Ten list on caching systems, we can borrow many of the well developed concepts used to study the performance of caching systems in our analysis. In particular, the concept of *Hit Rate (HR)*. Hit Rate is the most popular measure of the performance of a caching system. It is the number of times a referenced resource is in the cache. For example a Hit Rate of 60% indicates that six out of every ten requests to the cache found the resource being requested in the cache. For the analysis of the Top Ten list this would mean that six out of the ten subsystems that were in the Top Ten list had faults in them as predicted by the heuristic used to build the list. Thus, the higher the Hit Rate the better the prediction power of the heuristic and the usefulness of the Top Ten list, as managers aren't wasting resources on subsystems that are not susceptible to faults while missing other subsystems that are susceptible.

Unfortunately, using Hit Rate is not sufficient to measure the practical efficiency of the Top Ten list algorithms. Hit Rate only tells us if a subsystem that had a fault was in the Top Ten list or not. We hope to give managers enough advance warning time to react to the fault prediction. For example, if we have a 90% Hit Rate yet the subsystems that have faults are put in the Top Ten list just seconds or minutes before the fault is discovered in them, then such predictions although from a theoretical stand point are valid they are not practically useful. We would like to have a measure that is more practical, as managers require enough time to react to the proposed predictions. Hence, the time of adding a subsystem to the Top Ten list is important to obtain a more accurate measure of the performance of the Top Ten list. In contrast for web or file systems, the time of entry of a resource in the cache does not matter as long as the resource was found in the cache when requested. To overcome this limitation of the Hit Rate, we adopted two new metrics:

1. *Adjusted Hit Rate (AHR)*: The adjusted Hit Rate is a modified Hit Rate calculation which counts a hit only if the subsystem had been in the cache/Top Ten list for over 24 hours (other time limits are possible). For example we do not count a hit if the subsystem has been in the Top Ten list for just a couple of minutes. This will prevent us from inflating the performance of the heuristics used to build the list. In the rest of the paper we use the term Hit Rate (HR) to refer to AHR, unless otherwise noted.

2. *Average Prediction Age (APA)*: The Average Prediction Age calculates for each hit how long on average a subsystem has been in the cache/Top Ten. Although HR has been adjusted to account for predictions with a very short warning, we measure the APA to get a better idea of the age of predictions. For example, two heuristic may have similar HR but one heuristic predicts on average faults a week ahead of time whereas the other predicts them a full month ahead of time. A longer APA indicates a better performing heuristic for building the Top Ten list.

Using the HR and APA metrics, we proceed to evaluate various heuristics proposed in the following section.

3 HEURISTICS FOR THE TOP TEN LIST

Many heuristics can be used to build the Top Ten list, in particular, previous findings and observations from published literature in fault prediction can be used to create heuristics. For the purposes of this paper, we chose to use the following heuristics for their simplicity and intuitiveness. They are by no means a full listing of all possible heuristics instead they are some examples to validate our proposed Top Ten list approach:

Most Frequently Modified (MFM)

The Top Ten list contains the subsystems that were modified the most since the start of the project. The intuition behind this heuristic is that subsystems that are modified frequently tend over time to become disorganized. Also, many of the assumptions that were valid at one time have the tendency to no longer be valid as more features and modifications are performed on these subsystems. Eick *et al.* studied the concept of code decay [3]. Graves *et al.* showed that the number of modifications to a file is a good predictor of the fault potential of the file [6]. In other words, the more a subsystem is changed the higher the probability that it will contain faults.

This heuristic will tend to have a high APA as frequently modified subsystems will remain in the Top list for a long time. This may degrade the HR of this heuristic as it won't adapt to changes in the modification of files. For example, if in one release of an operating system all the work has concentrated on improving the memory manager and in the following release all the work focused on improving the file system, then the MFM heuristic msy still be affected by the modification counts of the previous release and will give out bad predictions. This limitation is a concern for any frequency based approach and is commonly refereed to in the literature of caching as the *cache pollution problem* [2]. To overcome this problem, heuristics that update the list based on a combination of the frequency and recency of a modification could be used.

Most Recently Modified (MRM)
The Top Ten list contains the subsystems that were recently modified. In contrast to the Top Ten list built using the MFM heuristic, the MRM Top Ten list is changing at a much higher rate as new files are modified continuously and are inserted in the Top Ten list. The intuition behind this heuristic is that subsystems that are modified recently are the ones most likely to have a fault in them. Finding faults in subsystems that were not modified for a long time is highly unlikely. In [6], Graves *et al.* showed that more recent changes contribute more to fault potential than older changes over time.

Most Frequently Fixed (MFF)
The Top Ten list contains the subsystems that have had the most faults in them since the beginning of the project. The intuition behind this heuristic is that subsystems that have had faults in them in the past will always tend to have faults in them in the future. Again this heuristic, like MFM suffers from the *cache pollution problem*.

Most Recently Fixed (MRF)
The Top Ten list contains the subsystems that had faults in them recently. The intuition behind this heuristic is that subsystems that had faults in them recently will tend to have more faults showing up in the future till most of the faults are found and fixed. In contrast, a Top Ten list built using the MFF will be a lot more stable than a list built using the MRF, as the subsystems in the list won't be changed as often.

The aforementioned heuristics represent a small sample of a variety of heuristics that can be used to build a Top Ten list. Conceptually, each heuristic can depend on one or a combination of the following characteristics of a software system.

1. *Recency*: The recency of modifications or fault fixes applied to the source code, such as MRM and MRF.
2. *Frequency*: The frequency of modifications or fault fixes applied to the source code, such as MFM and MFF.
3. *Size*: The size of subsystems or the size of modifications.
4. *Code Metrics*: The fault density, the cyclomatic complexity [8], or simply the LOC.
5. *Co-Modification*: Subsystems modified together will tend to have faults during similar times, for example.

We note that the problem of fault prediction has some characteristics that are different from classical caching literature, in particular:

- Whereas for file and web systems the number of possible resources to be cached is rather large, the number of subsystems that are analyzed for inclusion in the Top Ten list is limited, as managers have a limited number of resources to allocate to investigate the suggestions of the Top Ten list.
- Furthermore, CPU usage, algorithm complexity, and responsiveness of the caching heuristics are not a major issue due to the small number of subsystems that need to be analyzed. Also we expect the Top Ten list to be generated on a daily or weekly basis (or whenever needed) thus more complex and elaborate algorithms could be used to build the list. This is not possible in web and file system caching where the user expects an immediate and quick response.
- Finally, as pointed out earlier, a simple HR metric is not sufficient to measure the practical benefits of a heuristic, as managers require enough advance warning time to react to suggestions.

4 STUDIED SYSTEMS

To study the benefits of using the Top Ten list in the development of large software systems, we evaluated our proposed approach using six large open source software systems. In this section we give an overview of each of these systems. Table 1 summarizes the details for these software systems. The oldest system is over ten years old and the youngest system is five years old. For each system, we list the number of subsystems it has and the number of faults that were discovered in it according to our fault discovery process described below. For example, the Postgres database systems is written in C, contains 104 subsystems, and has had 1401 faults over its lifetime.

In the following subsections, we give details of the studied software systems. To measure the performance of the Top Ten list, we used the development history of these six software systems. The development history is stored in a source control system, such as CVS [13, 15] or Perforce [12]. The source control system stores all modifications that occur to each subsystem in the software system as it evolves. Each modification records the changed lines in the subsystem, the reason for the change, and the exact date of the change. Using a lexical technique, similar to [9], we automatically classify modifications into three types based on the content of the detailed message attached to a modification:

Fault Repairing modifications (FR): These are all modifications which contain terms such as *bug*, *fix*, or *repair* in the detailed message attached to the modification. The Top list attempts to predict ahead of time which subsystems are most susceptible to have such a modification applied to them in the near future.

General Maintenance modifications (GM): These are modifications that are mainly bookkeeping ones and do not reflect the implementation of a particular feature. These modifications are removed from our analysis and are never considered. For example, modifications to update the copyright notice at the top of each source file are ignored. Modifications that are re-indentation of the source code after being processed by a code beautifier pretty-printer are ignored as well.

Feature Introduction modifications (FI): These are modifications that are not FR or GM modifications.

The detailed description of the history of code development provides a rich opportunity to replay the history of the development of a software system and measure the benefits that the developers would have got if approaches such as the Top Ten list were accessible to them.

Application Name	Application Type	Start Date	Subsys. Count	Faults	Prog. Lang.
NetBSD	OS	21 March 1993	393	2451	C
FreeBSD	OS	12 June 1993	182	3264	C
OpenBSD	OS	18 Oct 1995	401	1015	C
Postgres	DBMS	9 July 1996	104	1401	C
KDE	Windowing System	13 April 1997	167	6665	C++
Koffice	Productivity Suite	18 April 1998	259	5223	C++

Table 1: Summary of the Studied Systems

Postgres DBMS

Postgres is a sophisticated open-source Object-Relational DBMS supporting most of the SQL constructs. Its development started in 1986 at the University of California at Berkeley as a research prototype. Since then it has become an open source software with a globally distributed development team. It is being developed by a community of companies and people co-operating to drive its development. In our case study we use data beginning with 1996 when Postgres became an open source project.

KDE K Desktop Environment

Another system we examined in our case study is the *KDE* (K Desktop Environment) system. The *KDE* project is an Open Source graphical desktop environment for Unix workstations. It seeks to fill the need for an easy to use desktop for Unix workstations, similar to the desktop environments found under Mac OS or Microsoft Windows. With several hundred developers working on it, it is consists of over 2.6 million lines of code at present.

KOffice Office Productivity Suite

The *KOffice* productivity suite is an integrated office suite for *KDE*, the K Desktop Environment. The full suite is developed by a community of software developers online under an open source license. It features a full set of applications which work together seamlessly to provide the best user experience possible. The list of applications are: *KWord* a word processor, *KSpread* a spreadsheet application, *KPresenter* a presentation program, *Kivio* a visio-style flowcharting application, *Karbon14* a vector drawing application, *Krita* a raster-based image manipulation program like Adobe Photoshop, *Kugar* a business reports generating tool, *KChart* a chart drawing tool, *KFormula* a powerful formula editor, and *Kexi* a small database similar to Microsoft Access.

FreeBSD Operating System

FreeBSD is an operation system (OS) for desktop and server applications. It features a high performance networking and file system which are able to sustain high loads. It is used in many Internet and Intranet servers. It is based on the 4.4BSD code which in turn is based on the AT&T BSD code. It is being developed under an open source licence with many developers worldwide working on it. In contrast to Linux where Linus Torvalds gets to choose which features to add or remove from the OS, *FreeBSD* development model revolves around a group of hundreds of individual programmers called the “Committers”. The Committers have the ability to make any change needed to the official *FreeBSD* source base at any time. The selection of Committers and resolution of disputes are handled by the *FreeBSD* Core Team. The Core Team acts like a board of directors. A similar model is followed by the *OpenBSD* and *NetBSD* projects.

OpenBSD Operating System

OpenBSD is another BSD based operating system which is developed through an open source licence. It focuses on security with the goal of creating the most secure operating system available. The development focuses on code auditing and on ensuring that each line in the code base in analyzed for security holes.

NetBSD Operating System

NetBSD is derived from 4.4BSD and 386BSD code base. It is being developed with a primary focus on creating an extremely portable and flexible OS. It runs on over 30 hardware platforms and provides a lot of flexibility to enable research and experimentation with many different types of hardware, and protocols.

We believe that the variety of development processes used, implementation programming languages, features, domain of the studied software systems ensures the generality of our results and their applicability to different software systems. Nevertheless, our results may only generalize to open source projects since we did not analyze commercial software systems. In the following sections we explain how we used the development history of the studied software systems in our analysis. Also, we present the performance of the heuristics presented in Section 3 against each of the studied software systems.

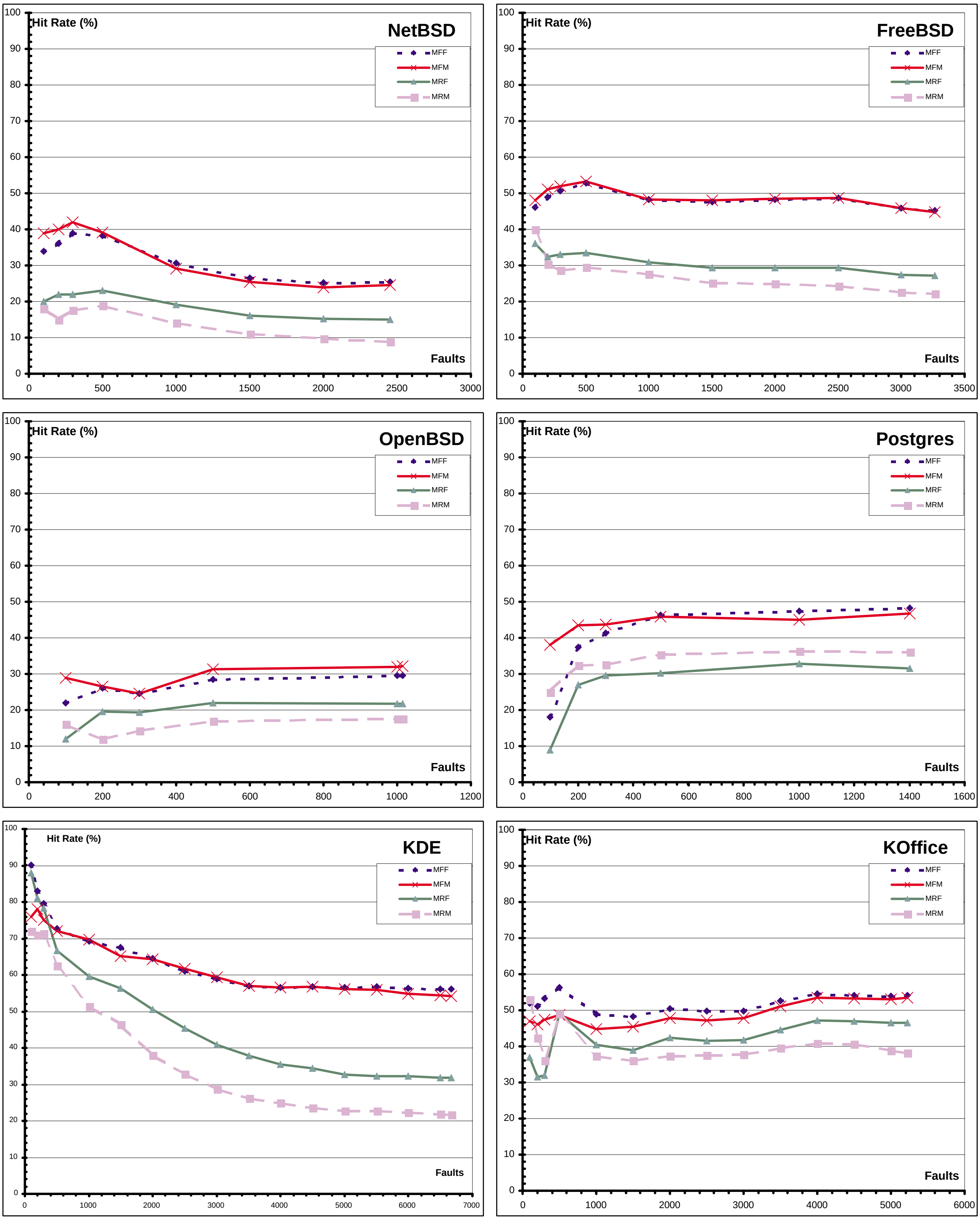

Figure 1: Hit Rate For The 4 Proposed Heuristics

5 MEASURING THE PERFORMANCE OF THE TOP TEN LIST

In this section, we measure the performance of the heuristics proposed in Section 3, to build the Top Ten list. For each of the software systems we analyzed the source control repository automatically without any user intervention. We chose to ignore the first year in the source control repository, due to the special startup nature of code development during that year as each project initializes its development process and the corresponding effect on its source code repository. We then used the following three years to measure the performance. For each heuristic, we plot the Hit Rate (HR) versus the fixed faults over the three year period. Furthermore, we calculate the total Hit Rate and the Average Prediction Age (APA) over the studied three years for each of the six open source systems we studied.

Figure 1 shows the performance for the four proposed heuristics. In the figure we show the *Hit Rate* of the Top Ten list using each heuristic for each fault that occurs. For example for *NetBSD* once there are 1000 faults, the Hit Rate for the heuristics are as follow: MFF (29%) MFM (30%) MRF (20%) and MRM (15%). We note that we do not show the Hit Rate for the first 100 faults, as we choose to use the first 100 faults to calibrate our Top Ten list with some historical data to gain a more realistic and fair comparison of the different heuristics as the Top Ten list fills up slowly over time.

Examining the figure, we note that the two heuristics (MFM and MFF) that are based on a count of modifications or faults have the best performance. In contrast, the other two heuristics (MRM and MRF) which are are based on the recency of modifications and detection of faults in a subsystem do not perform as well.

Furthermore, the performance of MFF at the beginning is always worse than the performance of MFM, this is due to the fact that at the beginning there are not as many faults thus the MFF heuristic performance is negatively affected. The need of MFF for a large number faults to calibrate itself suggests the need for a heuristic based on the modifications count at the beginning of the development of the project. Later on we may switch to a heuristic that is based on the fault counts if it is performing better. In our analysis, we see that around 400 - 500 faults, the MFF has enough faults to calibrate well.

Over time, the performance of the proposed heuristics either decline or stay constant except for the *Koffice* system where it improves. The decline in the prediction quality may suggest that the Top Ten list has been polluted by subsystems that were very highly modified/fixed in the past but are no longer being modified in the later years. An enhanced heuristic that overcomes this problem may be very beneficial in improving the performance of the list.

Table 2 summarizes the performance metrics over the three years of data used in the study. In particular, we notice that the unadjusted Hit Rate (HR) for the recency based heuristics, such as MRM and MRF, drops significantly once the Adjusted Hit Rate (AHR), which does not count hits with a short warning time, is calculated. By examining the Average Prediction Age we see that it is less than a day in many of the cases where the recency based heuristic is used.

Application	Heuristic	HR (%)	AHR (%)	APA (in days)
NetBSD	MRM	22.4	9	0.3
	MRF	20.6	15	0.8
	MFM	24.4	24.4	133.8
	MFF	25.3	25.3	138.7
FreeBSD	MRM	32.6	22.2	0.98
	MRF	32.6	27.2	1.7
	MFM	44.9	44.9	252.7
	MFF	45.1	45.1	245.1
OpenBSD	MRM	28.5	17.6	0.71
	MRF	24.5	21.8	3.11
	MFM	32.1	32.1	182.22
	MFF	28.8	28.8	168.5
Postgres	MRM	42.1	36.2	3.3
	MRF	35.4	31.4	4.4
	MFM	48.4	48.4	287.8
	MFF	46.6	46.6	288.6
KDE	MRM	46.6	21.7	1.4
	MRF	49.3	31.7	3.9
	MFM	54.3	54.3	375.4
	MFF	56.1	56.1	394.1
Koffice	MRM	53.6	38.3	2.4
	MRF	56	46.6	4.6
	MFM	53.4	53.4	133.8
	MFF	54.1	54.1	341.3

Table 2: HR, AHR, and APA for the Studied Systems During the 3 Years

6 THE EFFECTS OF A LARGER LIST

In the previous section, we presented the performance of the Top Ten list approach using various heuristics. In this section, we examine if increasing the size of the list would improve the performance of the heuristics. Due to space restrictions we will focus on only two of the four proposed heuristics, we chose MFM to represent the frequency based heuristics as its performance is very similar to MFF and we chose MRM to represent the recency based heuristics as its performance is similar to MRF.

For both MFM and MFF, we re-ran the same experiments done in the previous section while varying the size of the Top Ten list. We chose to make the size of the list as a function of the number of subsystems (subdirectories) in the software system. Thus we chose to have the size of the list vary between 2%, 10%, 20%, 50%, 80%, and 100% of the number of subsystems. In the case of 100%, we are able to see the best possible HR but unfortunately this is not practical as

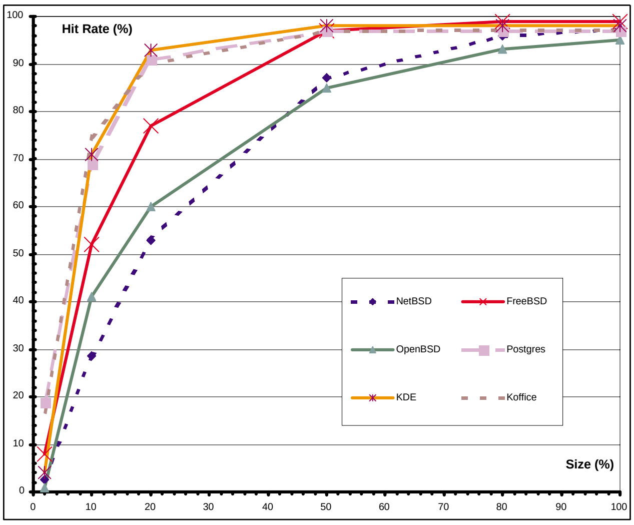

Figure 2: Hit Rate Growth As a Function of The Top List Size Using MRM Heuristic

managers would have to focus their attention to all the subsystems in the software system which defeats the purpose of a Top list.

Figures 2 and 3 show the growth of the Hit Rate as we vary the size of the Top list. We notice that when the Top list size is under 50% of subsystems in the software system then MFM (frequency based heuristic) outperforms the MRM (recency based heuristic). Once we are above 50% both types of heuristics have the same performance. Also we can never reach a Hit Rate of 100% as we always have misses in our predictions as we populate the list initially. For example, for the MFF heuristic a subsystem would have to have at least one fault that was not predicted at the beginning to be considered for inclusion in the predicted Top list.

Examining the growth of the Hit Rate in Figures 2 and 3, we notice that the Hit Rate exhibits a logarithm growth as we increase the size of the Top list. This indicates that the benefit of increasing the size of the Top list diminishes exponentially. From both figures, we see that a Top list which is around 20% the number of subsystems in the software achieves the best return on investment for managers. The 20% value supports previous findings by Munson and Khoshgoftaar [10] and by Adams [1] which showed that faults tend to occur in a subset of the subsystems of large software systems. It also showcases the value of using the Top Ten list as managers will expend less effort on fault-free subsystems and can allocate more resources to troublesome ones.

7 DISCUSSION

In this section, we elaborate on issues relating to the performance of the heuristics used to build the Top list.

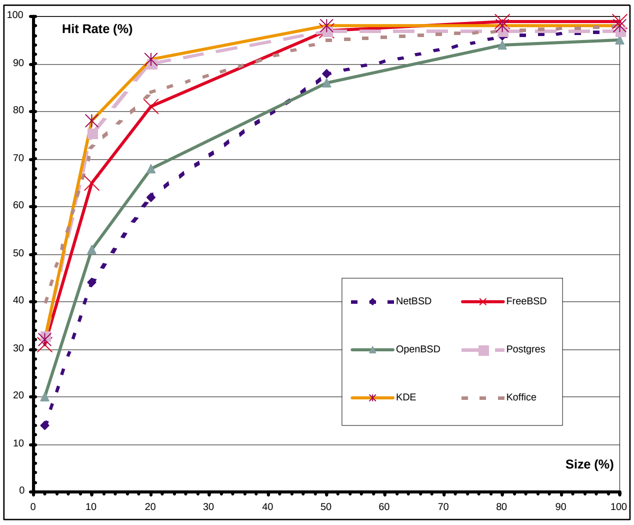

Figure 3: Hit Rate Growth As a Function of The Top List Size Using MFM Heuristic

An Accurate Measure of the Performance of a Heuristic

The Top Ten list assists managers in allocating testing resources by focusing on the subsystems that are likely to have a fault appear in them in the near future. In our analysis, we used the change history to measure the performance of heuristics. In particular, we used the fact that a fix was applied to a subsystem as an indication that a fault was detected in that subsystem. A record of all reported bugs from the field throughout the lifetime of a software system could have been used to evaluate the performance of the different heuristics. Unfortunately, for many open source systems detailed bug tracking systems do not exist. Furthermore, reported bugs are not an indication of the occurrence of faults or their severity, since the reported bugs may be due to misunderstandings of the functionality of the software system by the reporter of the bug. We believe that the use of fixes instead of bug reports is a reasonable measure since detailed bug reports do not exist for the systems we studied, and managers tend to focus their resources on the most critical faults.

Performance of Fault Based Heuristics

In our analysis we used two heuristics (MFF and MRF) that are based on fault counts. Unfortunately even though these two heuristics have good performance as presented in the previous section, it may be challenging to measure their performance if a Top list did actually exist for the development team. The Top list biases the effort and work performed by a development team. There is a high tendency for developers to focus their testing resources to subsystems that are in the Top list. Thus over time, the fault discovery may be influenced by the Top list and using the fault counts becomes an inaccurate measure. Instead using heuristics based on the modification counts are likely to be more stable and unaffected by the Top list suggestions. This poses an interesting challenge for software engineering research where introducing new techniques to a process may invalidate the analysis of benefits of the new techniques. Thus, even though historical data show the benefits of a research idea, validating the idea in a practical setting may reveal interesting challenges and issues.

Determining a Practical Average Prediction Age
Throughout the paper we emphasized the need for heuristics that are able to provide high HR. To ensure that our results are useful and practical we measured the Prediction Age (PA) for each hit and chose not to count hits with low PA. As a manager is not given enough warning to react when the PA is low. We then chose to measure the APA which is the sum of the PA's for all the Hits divided by the number of hits. Looking at Table 2, we list the APA for all heuristics for each of the studied software systems. As pointed out earlier, recency based heuristics have a rather low APA. Unfortunately, frequency based heuristics have a high APA. This is mainly due to the *cache pollution problem*. The need for a heuristic that can combine a low APA with a high HR is justified. It would be very useful and practical for managers to get advance warnings that are not too early and are not too late. We now briefly discuss and present some measurements for such a heuristic.

Based on the results shown in Table 2, we would like a heuristic which keeps track of the recency and measures the frequency of events as well. We propose the use of an exponential decay function to build our heuristic. The decay function would reduce exponentially the effect of a modification or a fault on the probability that a fault will be discovered, based on how long ago a fault/modification to the subsystem has occurred. Then to measure the frequency, instead of adding up the number of times a modification/-fault occurred, we add up the exponentially decayed values. Consequently, given two subsystems who both have had 3 modifications to them, the subsystem with the 3 more recent modifications will have a higher heuristic value and would be considered more likely to have a fault discovered in the near future. More formally, we define a heuristic function (HF) and the Top list is created by choosing subsystems with the highest HF value. The HF for a modification based heuristic is defined as:

$$HF(S) = \sum_{m \in M(S)} e^{T_m - Current\ Time}$$

where $M(S)$ is the set of modifications to a subsystem S and T_m is the time of modification m.

We reran our results on four of the software systems in our system. Table 3 shows the performance results for using an exponential decay heuristic. We note that the APA values are much more moderate compared to the corresponding values shown in Table 2. The APA suggests that the new heuristic provides enough early warning and is still capable of dynamically updating as the development of the project progresses over time.

Application	AHR (%)	APA (in days)
NetBSD	25.3	26.1
FreeBSD	42	129
OpenBSD	33.1	38.6
Postgres	49	33.8

Table 3: AHR and APA for the Exponential Decay Heuristic

8 RELATED WORK
The work most closely related to our work is done by Khoshgoftaar *et al.* In [7], they present a technique to predict the order of the subsystems that are most likely to have a large number of faults. The main similarity between our work is the recognition that managers have limited resources and need to focus these limited resources on a selected few subsystems in a large software project. Whereas, Khoshgoftaar orders subsystems based on their degree of fault proneness, we order subsystems based on their likelihood of containing a fault in the near future. Thus, our technique may choose to rank highly subsystems that may not be considered fault prone, yet they may have a fault appearing very soon in them.

9 CONCLUSIONS AND FUTURE WORK
We presented a new approach to assist managers in determining which subsystems to focus their limited resources on. By using this approach managers should be able to allocate testing resources wisely, locate faults in a timely manner and fix them as soon as possible. The approach uses ideas that have been extensively researched in the literature of web and file systems. The idea of caching as a limited resource is extended to the idea of limited testing resources. We show that the problem of determining which entities to cache is similar to the problem of determining which subsystems to focus testing resources on. We present the concept of Hit Rate which is widely used to measure the performance of various caching heuristics. Then we extend it to measure the performance of our heuristics that are used to build the Top Ten list.

We studied our proposed approach and heuristics using the development history of six large open source project. We saw that we can achieve a Hit Rate that is higher than 60% for some of the systems. We then examined the possibility of increasing the size of the Top Ten list and noticed that a list that contains 20% to 30% of the subsystems in a software system provides very good results even when using rather simple heuristics (Figures 2 and 3). We then presented a more elaborate heuristic based on an exponential decay function. We showed that the results using the new heuristics combine the benefits of early warnings for faults and the ability to dynamically adjust as new development data is available.

In this paper, we proposed the Adjusted Hit Rate (AHR) metric which counts warnings for which the managers had enough time to react and does not count hits with a short warning time. For example, a warning before a bug occurs by a day or two may not be counted but a warning before a bug occurs by a week or two would be counted. Mathematically speaking, the AHR metric uses a step function which assigns a value of zero if a warning is less than a day or two and one for all other warning times. It would be interesting to investigate other types of metrics to gauge the value of a heuristic. For example we could consider a metric which uses a bell curve shaped function to assign a low value for too early (minutes) or too late (months) warnings but assigns high value for warnings with enough adequate time (*e.g.* 2-3 weeks).

We believe that the Top list approach holds a lot of promise and value for software practitioners, it provides a simple and accurate technique to assist them in maintaining large evolving software systems.

ACKNOWLEDGEMENTS
The authors gratefully acknowledge the significant contributions from the members of the open source community who have given freely of their time to produce large software systems with rich and detailed source code repositories; and who assisted us in understanding and acquiring these valuable repositories. The anonymous reviewers gave helpful comments on earlier revisions of this paper

REFERENCES

[1] E. Adams. Optimizing preventive service of software products. *IBM Journal for Research and Development*, 28(1):3–14, 1984.

[2] W. Y. Chen, S. A. Mahlke, P. P. Chang, and W. mei W. Hwu. Data access microarchitectures for superscalar processors with compiler-assisted data prefetching. In *International Symposium on Microarchitecture*, pages 69–73, 1991.

[3] S. G. Eick, T. L. Graves, A. F. Karr, J. Marron, and A. Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 1990.

[4] S. G. Eick, C. R. Loader, M. D. Long, S. A. V. Wiel, and L. G. V. Jr. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering (ICSE 1992)*, pages 59–65, Melbourne, Australia, May 1992.

[5] N. E. Fenton and M. Neil. A Critique of Software Defect Predction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[6] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting Fault Incidence Using Software Change History. *IEEE Transactions of Software Engineering*, 26(7):653–661, 2000.

[7] T. M. Khoshgoftaar and E. B. Allen. Predicting the Order of Fault Prone Modules in Legacy Software. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pages 344–353, Paderborn, Germany, Nov. 1998.

[8] T. J. McCabe. A Complexity Measure. *IEEE Transactions Software Engineering*, 2(6):308–320, 1976.

[9] A. Mockus and L. G. Votta. Identifying Reasons for Software Change Using Historic Databases. In *Proceedings of the International Conference on Software Maintenance (ICSM 2000)*, pages 120–130, San Jose, California, Oct. 2000.

[10] J. Munson and T. Khoshgoftaar. The Detection of Fault-Prone Programs. *IEEE Transactions of Software Engineering*, 18(5):423–433, 1992.

[11] N. Ohlsson and H. Alberg. Predicting Fault-Prone Software Modules in Telephone Switches. *IEEE Transactions of Software Engineering*, 22(12):886–894, dec 1996.

[12] Perforce - The Fastest Software Configuration Management System. Available online at `<http://www.perforce.com>`

[13] M. J. Rochkind. The source code control system. 1(4):364–370, 1975.

[14] N. F. Schneidewind. Methodology for Validating Software Metrics. *IEEE Transactions on Software Engineering*, 18(5):410–442, May 1999.

[15] W. F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.