Software Defect Tracking during New Product Development of a Computer System

By

Lisa A. Curhan

M.S. Electrical Engineering, University of California at Berkeley, 1985
B.S. Electrical Engineering, Brown University, 1983

Submitted to the System Design and Management Program
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management

at the

Massachusetts Institute of Technology

February 2005

Signature of Author _____

Lisa A. Curhan
System Design and Management Program
February 2005

Certified by _____

Nancy Leveson
Thesis Supervisor
Professor of Aeronautics and Astronautics

ABSTRACT

Software defects (colloquially known as "bugs") have a major impact on the market acceptance and profitability of computer systems. Sun Microsystems markets both hardware and software for a wide variety of customer needs. The integration of hardware and software is a key core capability for Sun. Minimizing the quantity and impact of software defects on this integration during new product development is essential to execution of a timely and high-quality product.

To analyze the effect of software defects on the product development cycle for a midrange computer system, I have used a particular computer platform, the Product1 server, as a case study. The objective of this work was to use Sun's extensive database of software defects as a source for data-mining in order to draw conclusions about the types of software defects that tend to occur during new product development and early production ramp. I also interviewed key players on the Product1 development team for more insight into the causes and impacts of software defects for this platform.

Some of the major themes that resulted from this study include:

The impact of defects is not necessarily proportional to their quantity. Some types of defects have a much higher cost to fix due to customer impact, time needed to fix, or the wide distribution of the software in which they are embedded.

Software Requirements need to be vetted extensively before production of new code. This is especially critical for platform-specific requirements.

The confluence of new features, new software structure and new hardware can lead to a greater density of software defects. The higher number of defects associated with the new System Controller code supports this conclusion.

Current Limitations of Defect Data Mining: Automated extraction of information is most efficient when it can be applied to numbers and short text strings. However, the evaluation of software defects for root cause cannot be easily summarized in a few words or numbers. Therefore, an intelligent classification methodology for root causes of software defects, to be included in Sun's defect database, would be extremely useful to increase the utility of the database for institutional learning.

Software Defect Data Mining seems to be underutilized at Sun. I have barely touched the surface of the information that can be extracted from our "BugDB" defect database. This data resource is rich with history. We should extract and analyze this type of data frequently.

ACKNOWLEDGEMENTS

I have many people to thank for supporting this research.

Deb Dibenedetto supplied very helpful sample SQL extracts from the Product1 QA organization.
Tilmann Bruckhaus lent valuable advice when I was selecting this thesis topic.
Hamid Sepehrdad provided information on the size and makeup of the BugDB database.
Rachel Hurwitz, John Mulligan and Tim Terpstra provided background information on the SCFW system controller debug history.
Tarik Soydan provided insight into the OS kernel put-back process.
John Leone and John Bell provided Product1 development schedule information.
Mike Naka gave me information on Product1 defects discovered in Operations and much moral support.

Joan Cullinane, Steve Klosterman and Scott Mitchell need to be thanked profusely for their support of my SDM graduate work throughout the program. Scott and Steve patiently reviewed my thesis drafts and lent an experienced Sun-internal perspective to the work.

My thesis advisor, Professor Nancy Leveson, is thanked for her excellent feedback and thoughtful editing suggestions.

Finally, my husband Dana Curhan has my eternal gratitude for many, many hours when he took on more responsibility at home so that I could complete this work.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

1.1 Glossary of Acronyms, Abbreviations and Industry Terms

Bug A hardware or software defect.

BugDB Refers to Sun's internal database of hardware and software defects, along with associated user interfaces and reporting tools.

Product1 A Sun midrange computer server. This server has a rackable form-factor configurable with multiple CPU modules.

Driver In the context of this thesis, a driver refers to a piece of operating system software responsible for interfacing with a particular piece of hardware

Product2 These machines were slightly less complex machines compared to Product1. They are members of the same product family. In BugDB some firmware defects found on both the Product2 machines and Product1 were categorized as Product2 bugs rather than P-family bugs.

Field The end customer or sales and service representatives who work directly with the end customer.

P-family Refers to the platform family of servers and workstations which use the same CPU technology as the Product1 system. In the context of this thesis, P-family firmware is the common firmware tree for this platform family.

GA General Availability. The date at which a product is available to ship in large quantities to end customers.

SPARC™ Sun's brand name for its CPU chip architecture

SunSolve™ Sun's online customer support information and patch service

OBP OpenBoot™ Process. Firmware code that configures the system and manages system resets prior to the start of operating system control.

Operations The division of Sun Microsystems responsible for product manufacturing.

POST Power-On Self Test. A set of firmware diagnostic tests that run every time the system is powered-on or on demand before operating system boot.

Diag Diagnostic. A BugDB category. In the context of this paper, it refers to diagnostic test code which runs under the Operating System.

RR Revenue Release. The first date at which a product is shippable to end customers.

RFE Request for Enhancement

OS Operating System

QA Software and System Quality Assurance. This group performs both targeted functional testing and general system-level testing.

DiagProg The primary diagnostic test program available for manufacturing and customers. It runs under the operating system.

SCFW The System Controller firmware on the Product1 server.

1.2 Structure of Thesis

Section 1 contains background information needed to understand the analysis presented in the later chapters. It discusses the structure of the main data source used for the research (the "BugDB" database), the Product1 hardware platform used as a case study, and the process used for the development of the Product1 and its associated system software and firmware. Section 1 also describes the objective of the thesis research and has an overview of the approach employed to obtain research data.

Section 2 has a brief literature review relevant to the thesis topic. The literature review is limited to the topic of software defect data mining and does not include references to computer software development or new product introduction in general.

Section 3 reviews the research methodology in detail. It describes how data was extracted from the database and interviews. It also discusses the graphical and statistical tools employed.

Section 4 presents the data uncovered in this research. It provides some basic data analysis and presents major themes for discussion in section 5.

Section 5 presents the findings from the data analysis illustrated in section 4. It supports the points outlined in the Abstract.

Section 6 presents conclusions and recommendations based on the work presented in sections 4 and 5. It proposes some future work that should be conducted at Sun in the areas of software defect tracking, software development and defect data mining.

Product names and bug category names have been modified throughout this thesis to protect Sun confidential information.

1.3 Background

During my tenure at Sun Microsystems in the role of a New Product Engineer supporting the development of assembly and production test processes for new computer systems, the operations

teams have encountered some software and firmware defects while executing prototype builds. Sometimes these problems are resolved quickly. On occasion, they disrupt the new product introduction schedule to an unexpected degree. My experience working through software and firmware bugs during prototype builds inspired this research topic. The topic I chose was an analysis of software and firmware defects occurring during new product introduction at Sun.

Sun Microsystems is a major vendor of computer servers, workstations and thin clients. Sun supplies its own operating system, firmware for booting the system and handling low-level hardware interfaces, firmware and software diagnostic code, plus system management firmware for handling the systems environmental and user console functions. Sun also supplies many middleware and user applications, but the operating system, boot firmware, diagnostics, and system management firmware is the code exercised during routine production testing. I have therefore confined my research on software defects during new product introduction to these pieces of code. I have not considered hardware defects directly in this research, although often hardware defects do induce software and firmware work-around.

In the next three subsections I will describe the defect database, case study hardware platform, product team structure, and product development process making up the work environment for software and firmware development studied by this research.

1.3.1 The "BugDB" Database

Tracking of system defects is taken very seriously at Sun. We utilize a bug tracking database with comprehensive user interface and data extraction applications. The system is referred to as

"BugDB"[1] in this thesis. Most hardware, software, and firmware defects found on Sun products are recorded in BugDB. The database records what pieces of code and what hardware is affected by the bug, the bug status, and how it is fixed. Internal and external customer complaints against the bug (customer calls) and any customer escalations against the bug are also recorded in BugDB. I have used the BugDB database as the main source of information for this thesis.

BugDB is a large, complex database. It is presently about 22 GB in size. It contains over a million entries, and multiple change records against those entries. Because the database is so large, it was necessary to limit my research to bugs submitted against chosen system software and firmware categories, in a limited timeframe.

The BugDB database has recently been converted from Sybase™[2] to the Oracle®[3] database format. I have continued to use the original Sybase™ data because the scope of my data extraction did not extend past the changeover date. I will comment further on the database and my data extraction methods in Section 3.

1.3.2 The Product1 Computer System

Due to the size of the BugDB database, I also needed to restrict my research to a limited scope of hardware. I have used the Sun Product1 server as a case study for this research. All the software and firmware defects examined for this thesis were seen (although not always exclusively) on the Product1 server hardware during the time period from its first prototype build to approximately one year after the server achieved general availability to customers.

Product1 is considered a "midrange" computer server. It is used in many industrial and business applications. Often Product1 is employed as a web server or database server. The Product1 form factor is designed for use in a 19 inch rack. .

Major Features of a typical Product1 server configuration include:

- Multiple SPARC™ CPU chips
- Several GB of memory
- Internal hard disk drives
- Ethernet ports
- I/O expansion slots
- A system controller card powered by its own CPU
- Redundant power supplies

The base set of system software for Product1 includes the following pieces of code:

- Operating System
- OpenBoot™ firmware
- POST firmware diagnostic suite
- System Controller firmware (referred to as SCFW)
- The DiagProg diagnostic test suite.

These are the only pieces of firmware and software covered by this research. A block diagram showing the major interfaces between these pieces of software and firmware is shown in Fig 1.2.

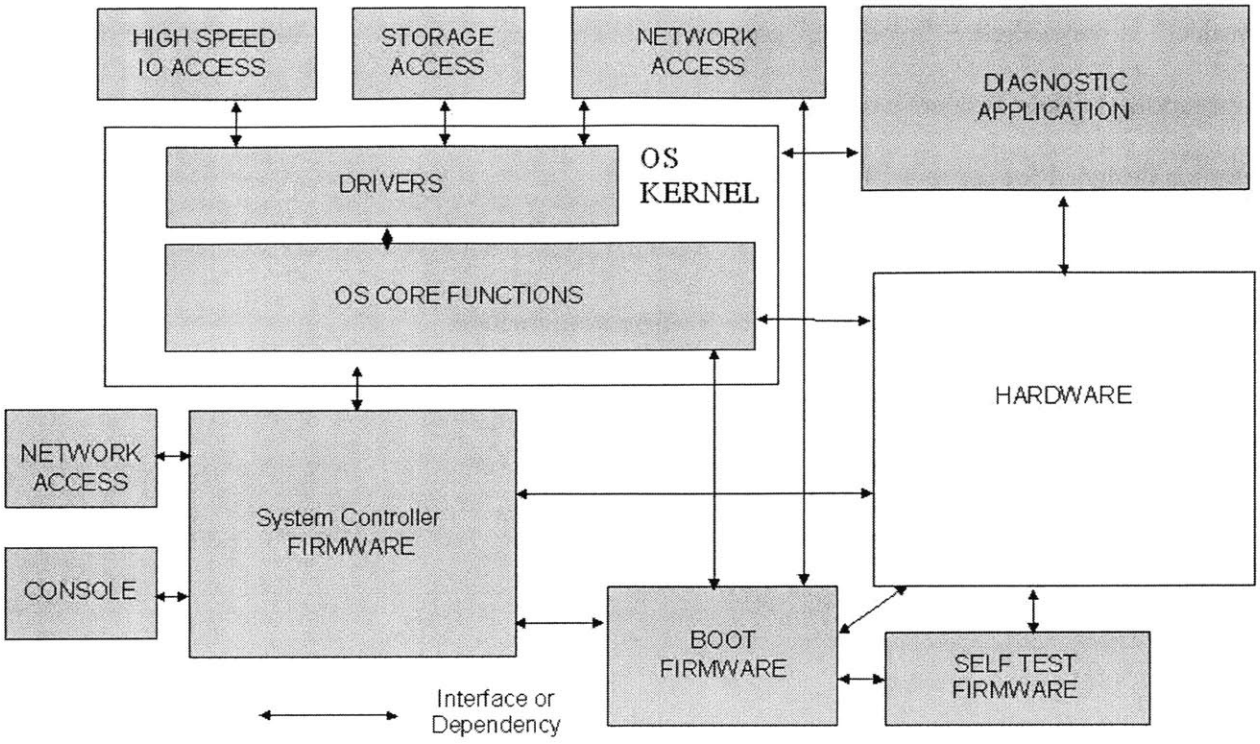

Figure 1.2: Product1 System Software Block Diagram

1.3.3 The Product1 Product Development Team

The structure of a typical Sun product development team is shown in Figure 1.3. The development team is led by a product Program Manager who may have one or more hardware and software product managers working under them. Developers of code or hardware report functionally to hardware or software group managers, but are also responsible to the product managers and product boss. Functions represented in the development team include Development, Marketing, Operations (manufacturing), Field Service and QA (Quality Assurance).

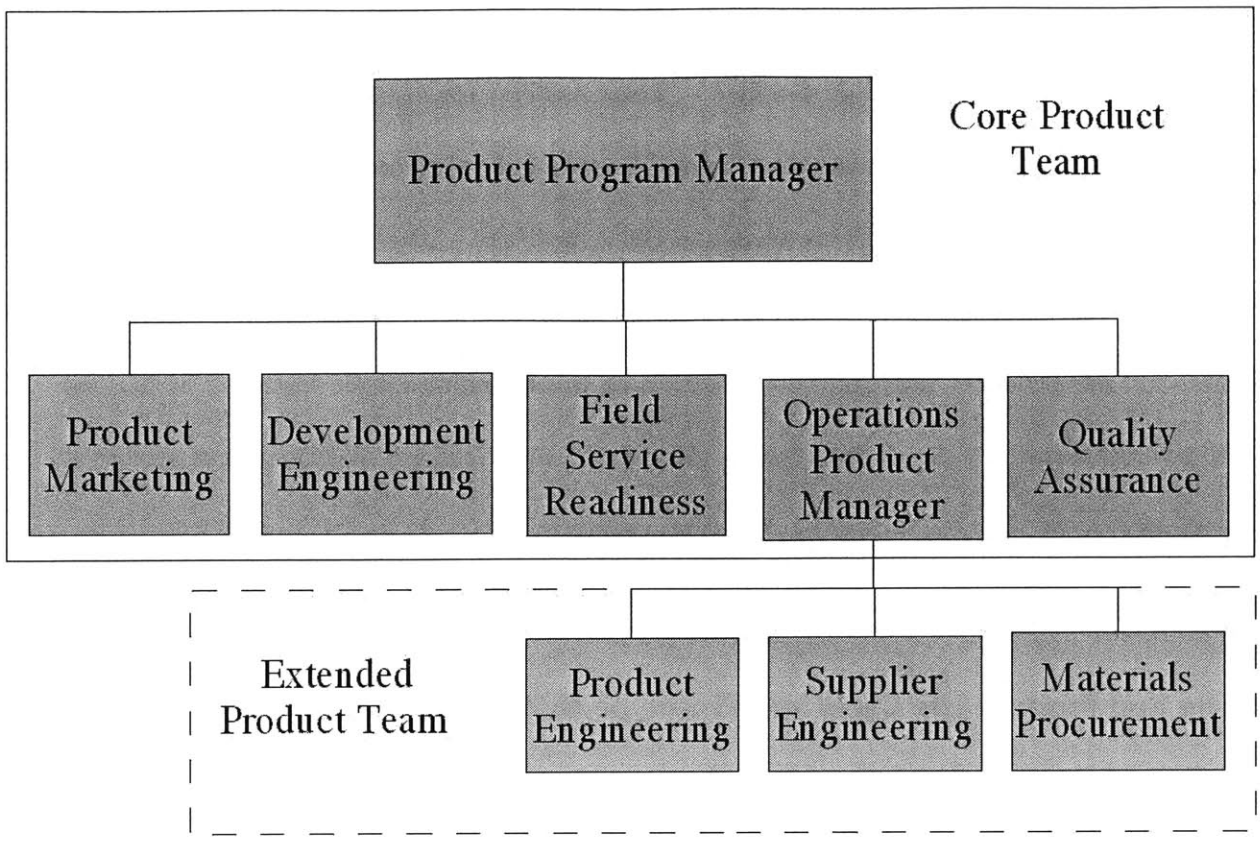

Figure 1.3: Typical Product Team Structure

Not all the hardware, software and firmware for a program are created by the development function on the product team. Many hardware and software components may be shared across a platform family or even across all Sun products. For example, the OS Kernel software is shared by all Sun products an OS supports for a given OS release. OS Kernel software is supplied by a central OS development group. Many of the extended software team members were geographically dispersed. A large portion of the OBP firmware related to the latest version of the CPU chip used in Product1 was supplied by a common source of "P-family" code written by a group located remote to the hardware design group. The POST firmware diagnostic code also had a common tree across the P-family product family, with some customized code for each platform. The POST firmware for the whole P-family product family was written by a group of developers who were co-resident with

the Product1 hardware team. The SCFW System Controller card firmware was also largely common across the P-family product family. Responsibility for System Controller firmware initially resided with a remote software team during Product1 development, and then was shared between a group local with the hardware team and the remote site. DiagProg diagnostic code was developed by a central diagnostic group located away from the rest of the team. Developers responsible for the Product1 platform customization of these common code bases act as liaisons from the product team to these shared support groups, and represent the shared support groups at team meetings.

1.3.4 The Sun Product Life Cycle and Prototype Builds

Sun shares a philosophy of phase-gate product development with many other organizations. Phases include Concept, Planning, Test, Launch, Ramp and EOL (end of life). The Test and Launch phases of the development process contain most of the prototype build and prototype testing activities.

The prototype builds for a given product are given sequential alphanumeric designations: P0, P1 and P2. Each build is preceded by design changes and fixes in the hardware and software. The final prototype build before production ramp-up is usually called the "Pilot" build, although sometimes P2 and Pilot were combined. The objective of the P0, P1 and P2.0 builds are generally fixed:

- P0: A small build for use by development only. Product functionality is demonstrated. Typically, a P0 build is tested only by the developer community.

- P1: A larger build in which Operations is involved in the build and test process. The QA group does its initial functional and systems testing on the P1 build. The P1 hardware is meant to be "feature complete". Subsequent design spins should improve or fix functionality, but not add features. The P1 build may also be a source of hardware and software for external Alpha or Beta testing.

- P2: The P2 build is primarily for the benefit of the manufacturing organization. It is meant to demonstrate volume manufacturing capability in the production assembly and test processes. The P2 build is usually the source for external Beta test machines.

Due to manufacturing, logistics, or design issues, the main prototype builds may be broken into smaller builds. These intermediate builds are quite common, and are usually planned. An intermediate build between P0 and P1 might be labeled P0.1 or P0.2. An intermediate build between P1 and P2 might be called P1.1 or P1.last.

1.4 Objectives

When I first started using the BugDB database to record bugs I encountered during prototype manufacturing builds, I was impressed by the amount of program history that was stored in its fields. I speculated that this database could be a source of organizational learning. There are now some functions within Sun that are using BugDB for organizational learning and process improvement[4], but for the most part BugDB is used for recording and tracking defects to make sure

that they are logged and acted upon properly. BugDB is also used as a source of metrics to assess development organizations on the quality of their processes.

Another common use for BugDB is as a reference for unexpected behaviors encountered during development or in the field. If the defect resides in hardware or software shared across platforms, often there is already a bug entry in the database that refers to it. The subsequent discoverers of the bug can then file a "customer call" field against the bug entry. Additional customer calls may increase the priority of the defect, and may provide additional information to aid developers in diagnosing the problem. Thus, the functionality of this database at Sun is mostly tactical. BugDB supports the day to day progress of the project.

The overall objective for this research was to use the data in BugDB in a strategic rather than a tactical manner so that information could be extracted to improve our firmware and software development processes. This objective can be broken down into three sub-objectives:

1. Expose what kind of software and firmware problems were more prevalent during new product introduction of the case study product.
2. Explore which functional groups within and outside the product team discovered various kinds of bugs and when they were discovered. This information could provide inferences about the effectiveness of the product testing.
3. Find out how long it took to correct defects found in the case study firmware and software. Look into what kind of defects took longest to fix and why.

1.5 General Approach

The general approach for this research was a five step process:

1. Extract relevant database fields from the "BugDB" database using conditional queries to restrict the information to the hardware and software case study areas.
2. Supplement the data extracted automatically with information gained from reading the text of selected bug reports and from interviewing product team members about selected topics and individual defect histories.
3. Format the automatic and manually extracted data into a manageable format for analysis.
4. Analyze the extracted data using graphical and statistical tools
5. Draw inferences and conclusions from the analysis.

More detail on the research methodology is contained in Section 3.

2.0 LITERATURE REVIEW

There is not yet a wide body of work available on data mining software defect repositories. However, the open software movement has provided wider access to such repositories, and interest is rapidly growing in using this data for software engineering process improvement and defect prediction.

In May of 2004 the first International Workshop on Mining Software Repositories[5] was held in conjunction with the 2004 International Conference on Software Engineering[6].

The Defect Analysis of this session presented some papers relevant to this thesis. In particular, Ostrand et al[7] proposed use of a negative binomial regression model whose independent variables are characteristics of the code files in a new software release. Among the variables used are the numbers of lines of code in the file, whether the file is new changed or unchanged, the number of previous releases, and the number of faults found in the previous release. The model, using these input variables, is able to predict which files have the highest probability of having the most faults. Ostrand's is a different kind of analysis than performed for this thesis, because this research only examined a snapshot of the defect database, rather than the evolution of the database along with the evolution of code characteristics. The Ostrand paper also discussed categorization of the change request based on the job function of the submitter, a topic covered in this work.

A paper submitted to the 2000 ICSE by Lezak et al[8] contained analysis of the root cause of defects contained in a modification request database. In particular, the focus on root cause categorization and customer-reported bugs has parallels with this work. The data representation chosen to display results (mostly bar charts) is also similar to this work. The categorization scheme selected in Lezak for root cause analysis is much more detailed than that used here, and a much larger set of defects was examined for root cause analysis. This was possible because the code chosen for examination was much smaller than that used in my case study, with the result that Lezak and his colleagues were very intimate with the details of the sampled defects.

Many authors have also focused on the implications of software re-use with respect to defect density. The 2004 MSR Workshop (referenced above) had several papers on this subject. Among these, the paper by Mohagheghi et al.[9] has many points relevant to this work. Mohagheghi et al. concluded that re-used components had a significantly lower defect density than non-reused ones. In addition, they concluded that re-used components tended to have a higher percentage of high-priority defects than non-reused components but a lower percentage of defects that were detected post-GA. This thesis supports Mohagheghi's conclusion that re-used components tend to have a lower overall defect density and a higher proportion of high-priority defects. However, it does not show that re-used components have a lower proportion of post-GA defects than newer code.

The impact of software requirements on defects is a third topic of interest. Javed, Maqsood and Durrani[10] performed data collection on software change requests (CRs) across several projects, along with data mining on a bug database similar to BugDB for the same projects. They found that the number of change requests is positively correlated with the number of defects for all levels of severity (or amount of change required) associated with the change request. However the highest-severity change requests, which require changes in the system design, have the most significant impact on the high-priority defects. This is intuitively logical, as is the conclusion that the initiation of change requests in the later phases of development (e.g. during testing rather than design or coding) led to a greater percentage of high-severity defects than those initiated during the early requirements and design phases. Interviews with engineers and managers involved in this case study also pointed to late-phase change requests as causal to defects. In fact, misinterpreted or unclear requirements were anecdotally mentioned in conjunction with some of the defects which led to customer escalations in this study.

Customer escalations were the subject of relevant research by my Sun colleague Tillman Bruckhaus[11]. Bruckhaus, et al, have used data mined from the Sun BugDB database, along with a customer escalation database, to build historical models using bug characteristics in order to predict future escalations. They have trained and validated a complex neural network model using this historical data. So far the model has been able to output a "risk level" for escalations that was validated against three months of data. The highest risk level (level 6) in the model output showed an escalation probability of 60%. This model is now being exercised against live data on a large software product in order to focus efforts on the highest-risk defects.

Finally, the project environment for a large-scale software development case study was the subject of a substantial paper by Perry, Siy and Votta[12]. They found that parallel development by separate teams of developers for interacting modules is correlated to quality problems. They studied the change management records of a large project in detail, and associated the defect density over a period of time for various code files with the maximum number of parallel changes per day over a period of ten years for the same file. They speculated that interfering changes, especially when changes overlap in time, can contribute to the defect density of a module being worked on in parallel by different developers. The issue of parallel development is also relevant to the anecdotal findings in this case study.

3.0 METHODS

This section contains the detailed description of the research steps outlined in section 1.5. We will review the tools that were used, and the limitations of this method.

3.1 Interviews

I conducted informal telephone interviews with three members of the team that developed and tested the SCFW (System Controller) firmware for the P-family series of products. I was able to talk to the former manager of the SCFW group, one of the Quality Assurance testers who worked extensively with that code, and one of the lead developers. I chose to talk to this group because that software module exhibited the highest defect density during the new product introduction phase for Product1. I also talked with a former Operations program manager for Product1 with respect to the program's schedule challenges, and with two of the new product engineers for the program. Other background information with respect to Product1 was obtained from my own experience with that product during my participation on some problem-solving "tiger teams" that were formed during the P1.0 through P2.0 phases of the prototype builds.

3.2 Data Extraction from BugDB

The Sun BugDB database user interface makes several data extraction and reporting tools available. Reports were generated using these tools as an initial exploration of the database. However, the standard reporting tools available from the user interface were not adequate because they lacked the capability to extract defect record data fields based on the hardware platform against which were

filed. (Some Requests for Enhancements will be filed against the BugDB reporting tools as a result of this work). As a result, it was determined that SQL (Structured Query Language) queries needed to be composed to obtain the reporting flexibility required.

The Sybase™ version of the database that I used was compatible with the ISQL[13] version of the query language. I obtained some sample queries from an engineer who had worked on the Product1 Quality Assurance team. These queries consisted of Solaris™ Korn Shell[14] (ksh) executable files containing Solaris™ commands that called the SQLplus query tool on the Sybase™ version of the Sun BugDB database and passed the appropriate SQL language commands to extract requested data. The QA engineer also kindly provided the database SQL access login used by her group.

I needed to modify the sample queries significantly to obtain the data needed. An example of one of the modified queries is shown in Figure A1 of the Appendices. The output of the queries was "piped" (re-directed) to a text file with the output data fields delineated by the "|" character. The data fields extracted directly from the database are summarized with definitions in Table 3.2.1.

DATA FIELD	DEFINITION
Priority	Priority rating set by submitter of bug record (1=highest, 5 = lowest)
Severity	Severity rating set by submitter of bug record
Category	Major category for the bug record
Bugid	Numeric string which uniquely identifies the bug record
Subcategory	Minor category for the bug record
Synopsis	Short text description of the bug or request for enhancement
Bugrfe	Three-letter field which defines whether the bug is a defect (bug), request for enhancement (rfe) or ease of use (eou) change request.
Sub_date	Date the bug was submitted into the system
Int_date	Date when the fix for the bug was integrated back into the product source base
User_type	Field for the type of user who submitted the bug: I (Internal), E (External), O (OEM)
User_role	Job function of the user who submitted the bug. These are usually Development, Functional Test, System Test or Technical Support (Field Service).
Cust_name	Name of customer contact (internal or external) for bug fix. The submitter of the bug will be the initial customer. Additional customers may add a customer call record, and their names, to the bug record as they discover the bug in their context.
Duplicate	Will contain the bugid the bug is a duplicate of. If the bug is not a duplicate, this field will be NULL
Escalation_num	If the bug was subject to customer escalation or escalations, this field will contain the unique numeric id number for the escalation(s).

Table 3.2.1: Data Fields extracted directly from the BugDB database

The SQL queries underwent several versions as I spoke to people from the Product1 software team and learned more about the BugDB categories related to the product. The original query supplied by the Product1 QA engineer was meant to create a weekly report for the Product1 product software team to monitor progress with respect to bug creation and fixes during development. Because the query was specific to the Product1 team, it included categories for the Product1 product

and the P-family product family, but not for the OS Kernel bugs and the DiagProg diagnostic application bugs which also affected the new product development. I expanded the query to include these Sun-wide categories, but limited the extraction by targeting only those bugs with customer calls on the Product1 hardware platform. Luckily the customer call table in the BugDB schema included a mandatory field called "hardware version", which identified the hardware platform the customer had detected the bug on. I did not extract this field, but used it extensively for filtering.

The categorization of Product1 bugs was subject to a few historical anomalies due to the evolution of the product team and the BugDB categories. For example, the category-subcategory pair "firmware-Product1", needed to be extracted because some early OBP and POST bugs were submitted that way instead of under the later "P-family-OBP" or "P-family-POST" categories. A few of the P-family firmware bugs were also submitted under the platform-specific "Product1" or "Product2" categories instead of "P-family" even though they were seen (had customer calls) on both the Product1 and Product2/Product3 hardware platforms. I worked around these anomalies with additional SQL queries.

3.3 Data manipulation and formatting

Several separate queries were written instead of one large query so that the resulting ASCII data files were a manageable size. The text files were imported into spreadsheets so that they could undergo some intermediate manipulation and be merged into one large spreadsheet. Some derived fields were then added to the extracted dataset while the data set was in spreadsheet format. The derived fields are shown in table 3.3.1:

DATA FIELD	DEFINITION
Int_Time	The number of days between bug submission and integration of the bug fix.
Log(Int_Time)	Log base 10 of the integration time.
Num_Calls	Total number of customer calls filed against the bug record.
Num_Esc	Total number of customer escalations filed against the bug record.
Build	The hardware prototype build timeframe when the bug was submitted. Values include P0, P0.1, P0.2, P1.0, P1.1, P1.last, P2.0, RR (Revenue release to GA), GA (General Availability to GA+ 1 year).
Problem	Short text summary of the basic problem type described in the bug evaluation
Root Cause	Short text summary of the root cause of the bug problem. This was derived by the thesis author based on reading the text of the bug comments and evaluation.

Table 3.3.1: Data Fields derived or calculated from BugDB database fields

To calculate the number of days that passed between bug submission and fix integration, the "DAYS" function of the Star Office™ spreadsheet was used:

Int_Time = DAYS (Int_date;Sub_date);

If a fix had never been integrated into the production code, a large negative number would be returned. This was useful for filtering purposes. I was able to filter out all rows that contained duplicate bugs or rejected bugs (those deemed "not a bug") by eliminating the rows where Int_Time < 0. Most of the analysis in section 4 is filtered to eliminate duplicate and rejected bugs in this manner. The Integration time analysis is also filtered to eliminate bugs with Int_Time = 0. This

invalid value was input by developers who did not submit bugs until they already had the fix in place, thus making the Sub_date field invalid as a date of bug discovery.

The Num_calls and Num_escalations columns were derived by adding the customer contact names associated with each bug record. The Build field was manually entered after sorting the dataset spreadsheet by submission date. A bug record was labeled with prototype build identification if it was submitted on or after the start of that hardware prototype build and before the start of the subsequent prototype build. Records were labeled with the "RR" (Revenue Release) identifier if they were submitted on or after the RR date but before the "GA" (General Availability) date. Bug records submitted on and up to one year after the GA date were labeled with the GA identifier.

The "Problem" and "Root Cause" fields were only added to the dataset for the bug records that either escaped the standard development and QA test processes, or those that were not integrated before being escalated by an external customer. These were bugs that were either discovered by Operations in production testing, submitted by Field Service on behalf of external customers, or associated with a customer escalation number. The Problem and Root Cause fields were manually derived by interpreting the synopsis, comments and evaluation text associated with the bug record. It would have been preferable to derive these fields for all the bug records in the dataset, however time limitations prohibited this. It might be possible to pursue this option using a statistically valid sample across the categories and subcategories of the dataset as done in Lezak et al[15] or using automated text data mining as suggested by Kreyss et al[16]

I had planned to also filter out RFEs (Requests for Enhancements) from the analysis sample. Figure 3.2.1 shows about 5% of the extracted bug records were categorized by the submitters as RFEs.

However, examination all the "RFEs" in the dataset led me to reject many of the submitters' categorizations. Some of the RFEs can be interpreted as defects because they involved original product requirements that were not met, or met incorrectly, in the code. On the other hand, some records characterized as "bugs" in the dataset were high-priority RFEs. It was not practical to manually examine all the text fields in every bug record in order to correct the bug/RFE field, so the RFEs were left in the analysis dataset. Elimination or retention of the RFEs during analysis did not alter the qualitative results.

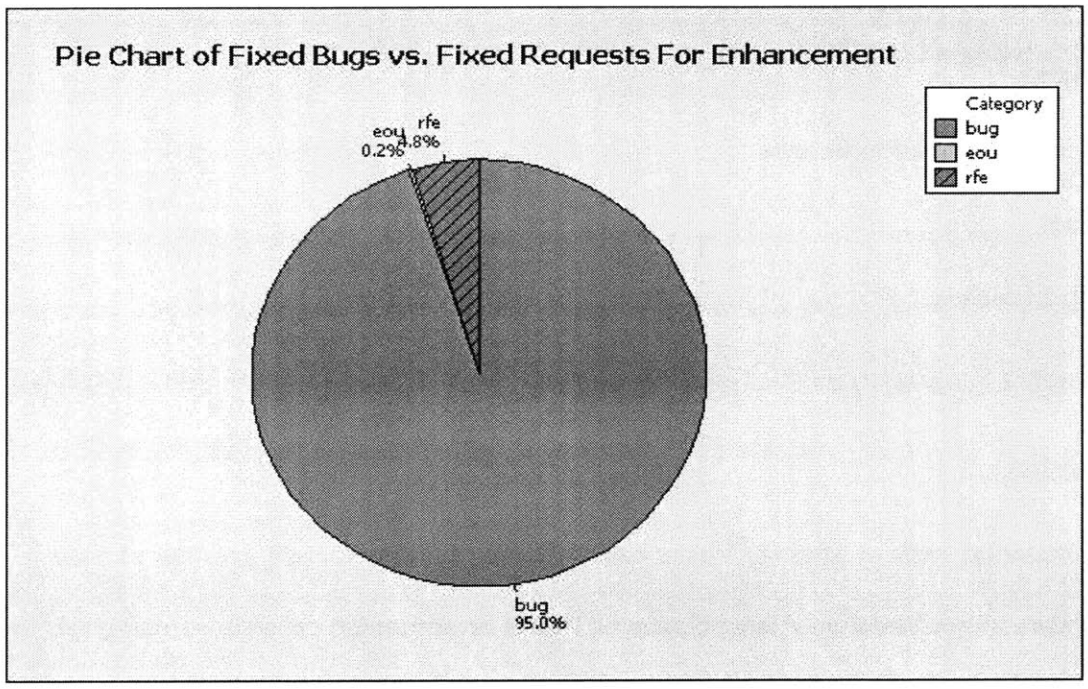

Figure 3.3.1: Percent Bug records in extracted data categorized as bug, rfe, or eou (enhancement of usability)

Once addition of derived fields was complete, the resulting spreadsheets were imported into my chosen statistical analysis tool, Minitab™.[17] In the next section I will review the methods used to analyze the extracted dataset, leveraging the Minitab™ software.

3.4 Data Analysis Methods

The Minitab™ statistical analysis software package is a flexible tool with both graphical output and tabular statistical output. Most of the analysis performed for this research is graphical analysis. Statistical analysis was performed on the available continuous variables. The graphical and statistical tools utilized were:

- Algorithmic data filtering
- Bar Charts (Pareto Charts)
- Histograms
- Cumulative Distribution Graphs
- Box Plots
- Analysis of Means

3.4.1 Data Filtering

All of the graph-generation tools in Minitab™ have a data-filtering option whereby you can include or exclude cells in the dataset based on a large choice of logical or arithmetic expressions using the numerical and categorical variables in the dataset. As mentioned previously, most of the analysis was filtered to eliminate duplicate and lowest-priority bugs by limiting the dataset to those bugs that had fixes put back into production code using the "Integration time > 0" condition. I also used the filtering options to examine bugs filed by certain functions, bugs discovered outside of Sun, and bugs implicated in customer escalations.

3.4.1 Bar Charts

Most of the analysis involving categorical variables such as the code module in which the bug occurred is presented in the form of bar charts or Pareto charts. The Minitab™ analysis package allows one to present the effects of using two or more categorical variables at once by generating stacked or grouped bar charts. For data confidentiality reasons, all the bar charts in this thesis are scaled as percent of the total bug count in the dataset.

3.4.2 Histograms

Histograms are used to present the distribution for the bug fix integration time variable, using both the actual data and the log-transform of the data.

3.4.3 Cumulative Distribution Graphs

Minitab™ supplies an automatic tool for comparing goodness of fit between empirical distributions and several common modeled distributions. This tool plots the empirical data's cumulative distribution on a scale that would make it linear it if it were a member of the modeled distribution's family. A correlation coefficient between the empirical data and the modeled line can then be calculated as a measure of fit. I used this distribution data-fitting tool to find a best-fit model for the distribution of the time between bug submissions and fix integrations.

3.4.4 Box Plots

Box plots are a graphical tool that compresses the representation of a distribution of continuous values by drawing a box containing the 25% to 75% (Inter-quartile) range, with a horizontal line indicating the median and straight-line tails stretching a further 1.5 times the inter-quartile range. Outliers, if any, are indicated with asterisk. Figure 3.4.4.1 illustrates these points.

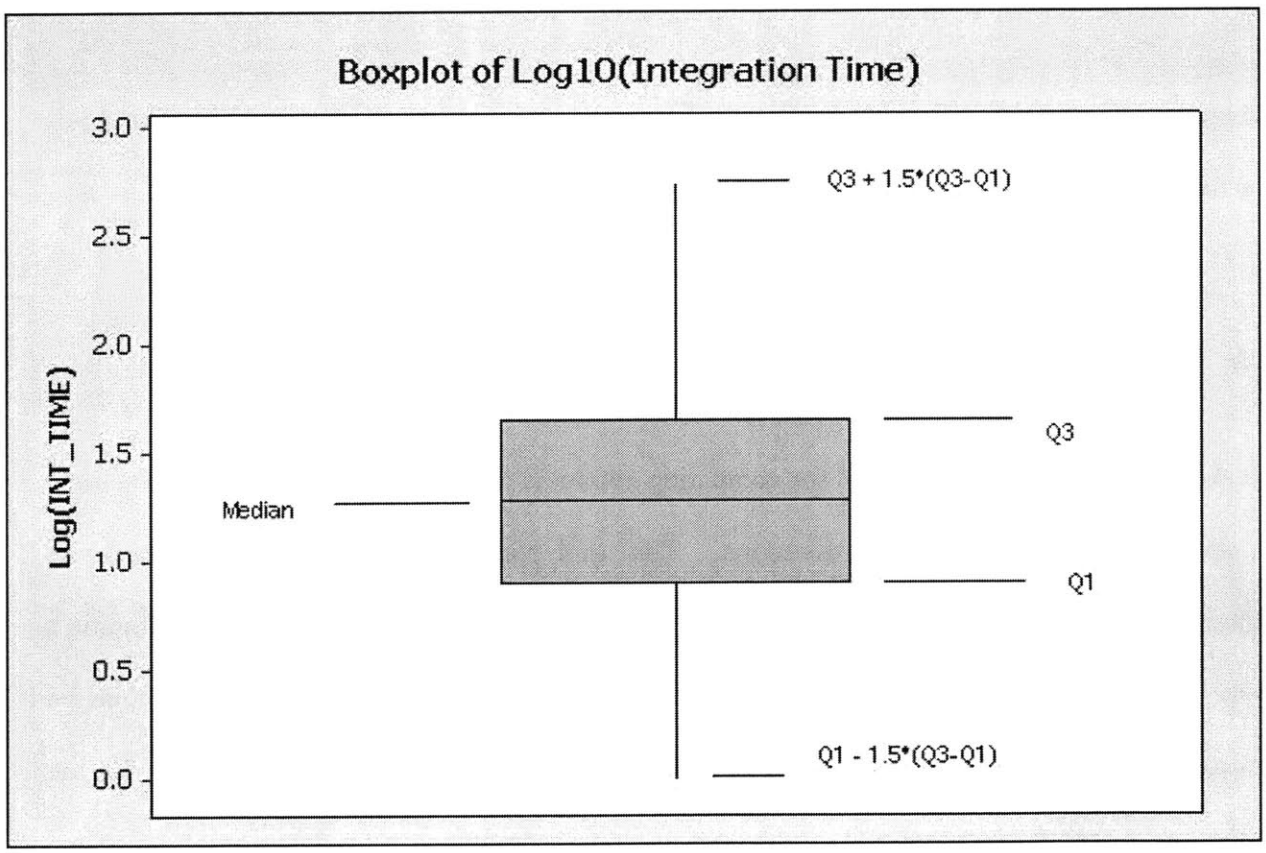

Figure 3.4.4.1: Example of a box plot

3.4.5 Analysis of Means

To determine whether two continuous distributions are different in a statistically significant way, one can perform an analysis of means. Minitab™ uses a version of the "T-test"[18] to prove or disprove the hypothesis (H0) that the mean of a distribution is the same as a comparison value. The results are presented graphically, with statistical confidence limits for H0 presented along with the actual population mean. When several distributions are being compared, the comparison value will be the calculated overall or "grand mean" of all the distributions. If the population mean for the distribution falls outside the statistical confidence limits for the null hypothesis (H0: distribution mean = grand mean) then the distribution does have a statistically different mean than the others to which others it is being compared. The larger the number of points in the distribution, the narrower the calculated confidence limits will be. Figures 4.4.6 and 4.4.7 are Analysis of Means graphs generated by the Minitab™ tool. Statistically different means are highlighted with a square outside the confidence limits. The distributions analyzed are assumed to be normal by the ANOM tool, so I used the normalized Log10 version of the integration time distribution. The default alpha value is 0.05, which results in a 95% confidence limit.

The next section, Section 4.0, will use the described tools and methods to present the results of the data analysis work.

4.0 RESULTS

4.1 When Bugs were found during New Product Introduction

The distribution of software and firmware bugs among the Product1 prototype builds followed expected trends. Most bugs were discovered (38%) during the P1.0 build, when the code first met all its functionality requirements. Many bugs (32%) were also found during the P0.2 build when substantial hardware was first available, and in the intermediate P1.1 build. Fewer bugs were found (around15%) after the P1.1 build. Figure 4.1 shows this distribution. In figure 4.1.1 the GA category includes bugs found up to one year after General Availability of the system.

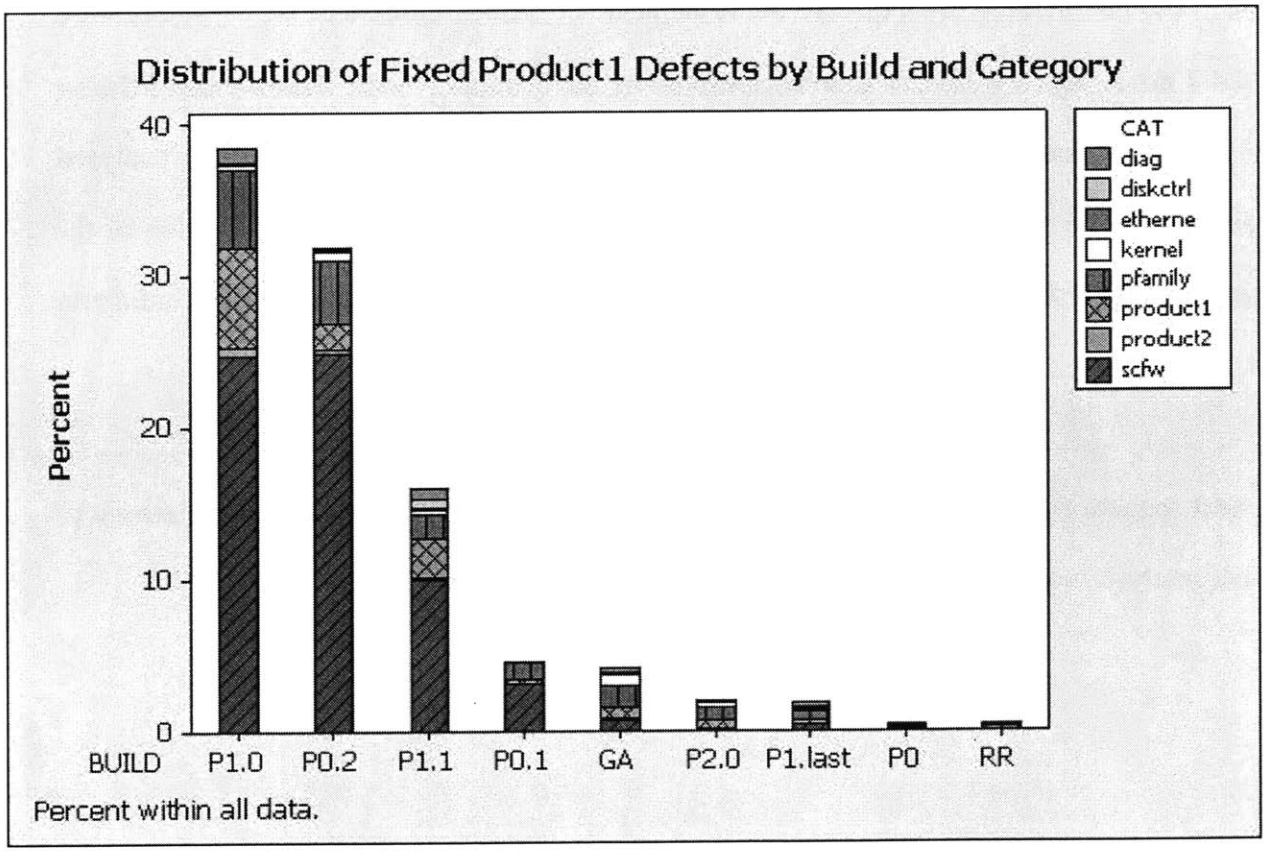

Figure 4.1.1: Distribution of defects by prototype milestone and category

4.2 Category/Sub-Category Analysis

The largest category of fixed bugs was for the SCFW (System Controller) firmware, which accounted for 63% of fixed Product1 system software bugs in this timeframe. Reasons for the proliferation of SCFW bugs are discussed in the next section. The next most prolific category was the P-family platform-family firmware, at about 15%, followed by Product1-specific firmware and Product1-specific OS code at about 13%. Figure 4.2.1 shows the distribution of defects by BugDB category.

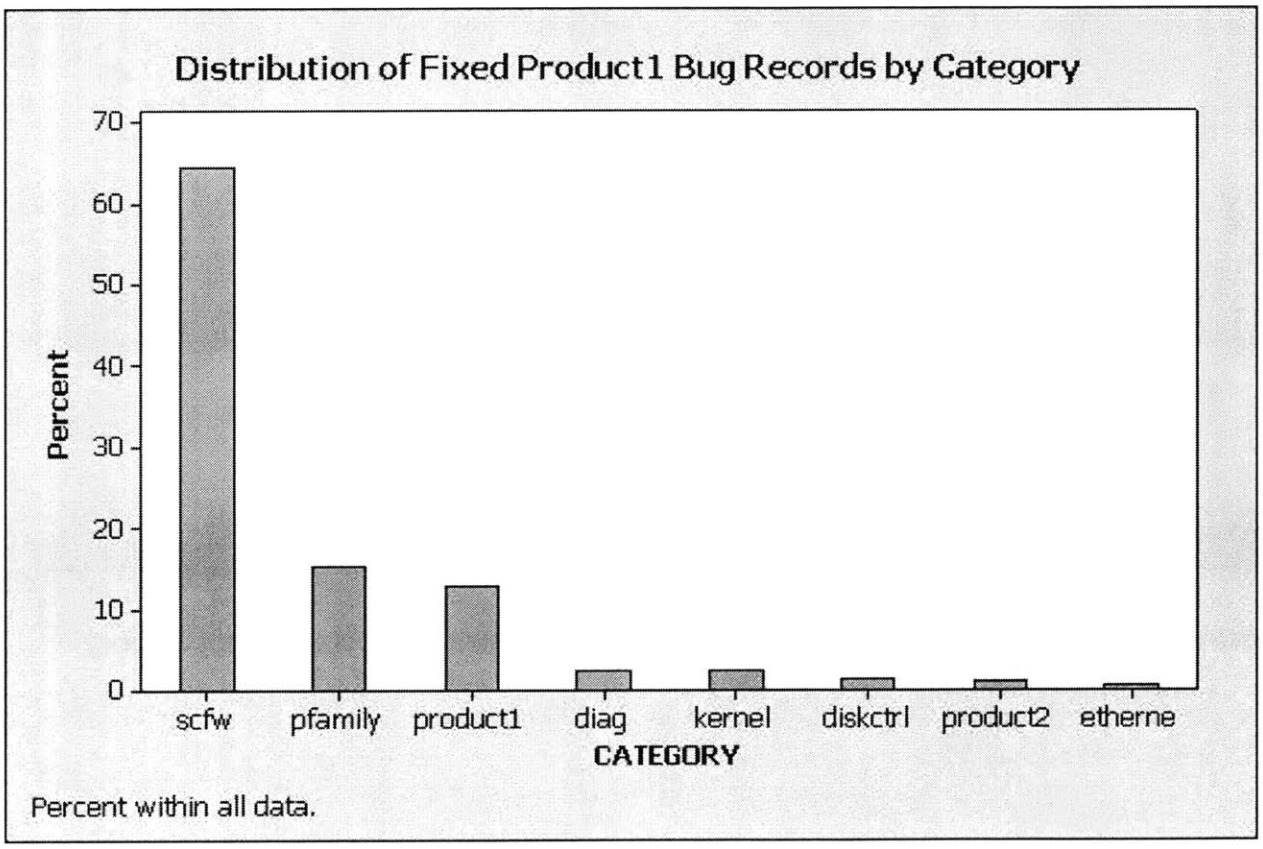

Figure 4.2.1: Distribution of Fixed Product1 System Software Defects by Category

The largest subcategories were the unfortunately-named SCFW "firmware_other" categories at 34% of bugs fixed, and the SCFW "firmware_environmental" category at 19% of bugs fixed. OBP (boot

code) firmware for both the Product1-specific and P-family platform code was next at about 13% of bugs. P-family POST and Product1-specific OS code bugs followed at 9% and 4% respectively. Figure 4.2.2 illustrates the distribution of fixed Product1 system software bugs in this timeframe by BugDB subcategory.

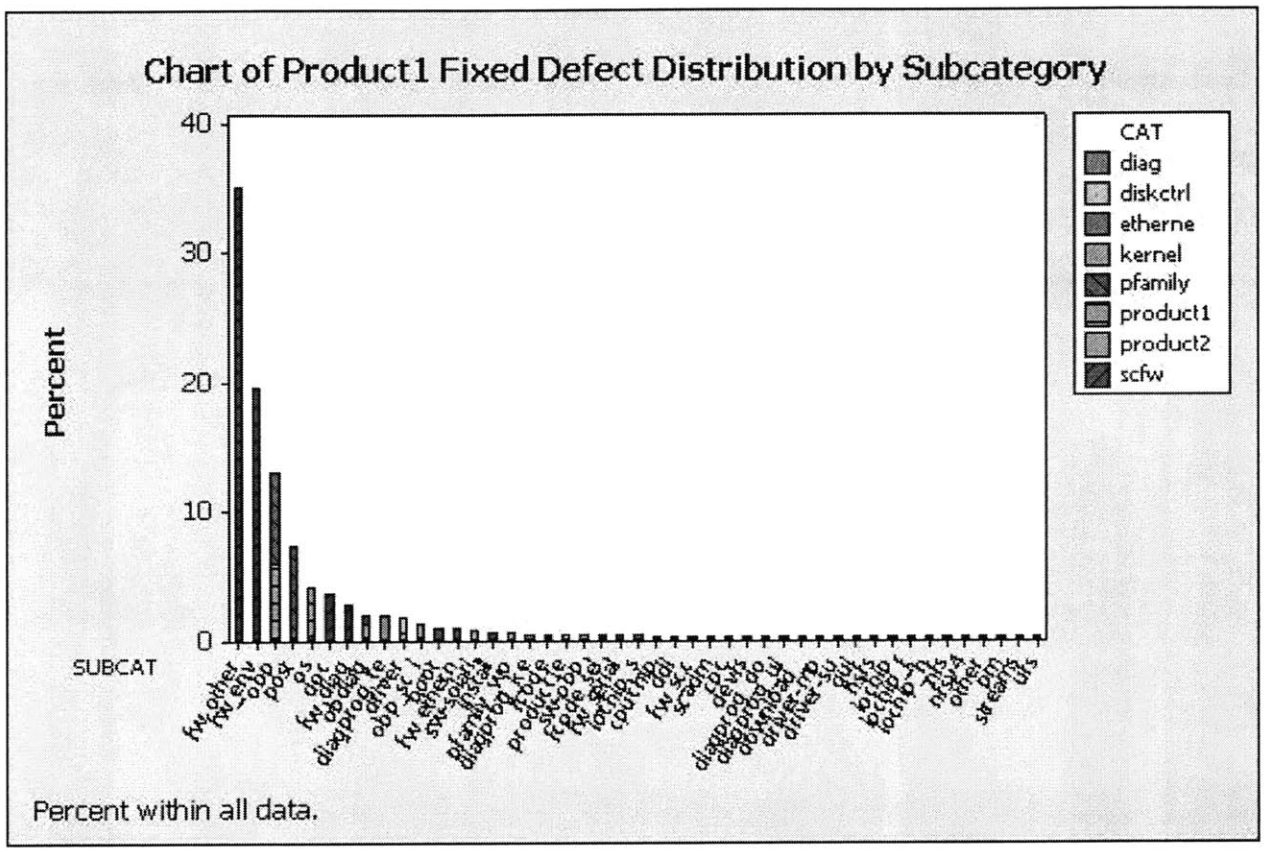

Figure 4.2.2: Distribution of Fixed Product1 System Software defects by Subcategory

Bugs discovered external to Sun (by Customers or Field Service) were not predominantly SCFW bugs. The biggest categories were P-family OBP and POST code at 41%, followed by SCFW environmental and "other" code at 25% and OS Kernel code in the pm and ddi modules at 16%.

Product1-specific OBP code accounted for about 7% of bugs found in the field and DiagProg diagnostic bugs accounted for another 7%. Figure 4.2.3 shows the distribution of this dataset of bugs discovered external to Sun. Notably, the order of categories in this distribution was very close to the order of categories in the bugs discovered after Product1 GA (General Availability).

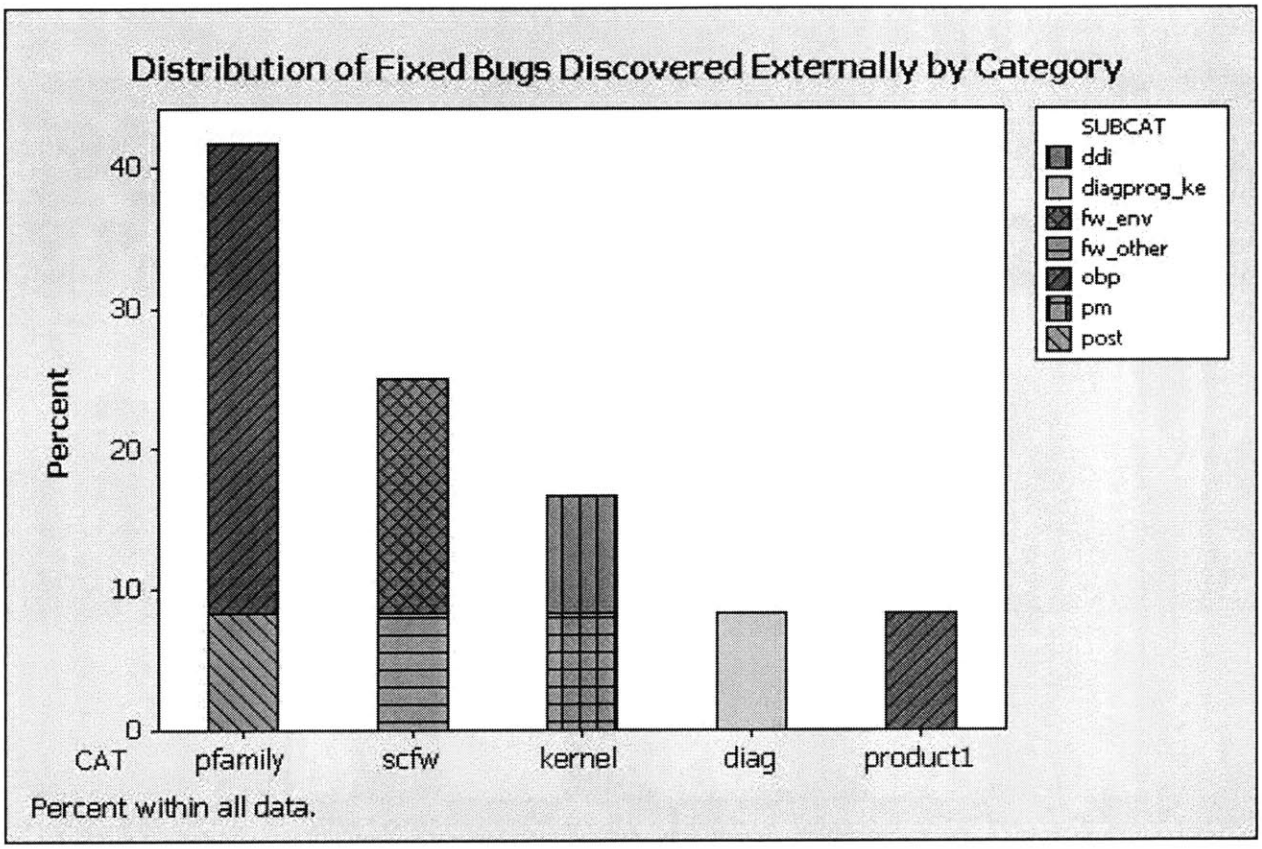

Figure 4.2.3: Distribution of Product1 system software bugs discovered externally

The bugs found by Product1 Operations reflect the production-testing environment. The largest percentage of bugs found by Operations (43%) was in Product1-specific OBP and OS code. 27% of the bugs filed in Operations were in SCFW diagnostic or environmental code. The remaining bugs were filed on DiagProg diagnostic code and P-family POST code. Figure 4.2.4 illustrates the distribution of bugs found by the Operations group.

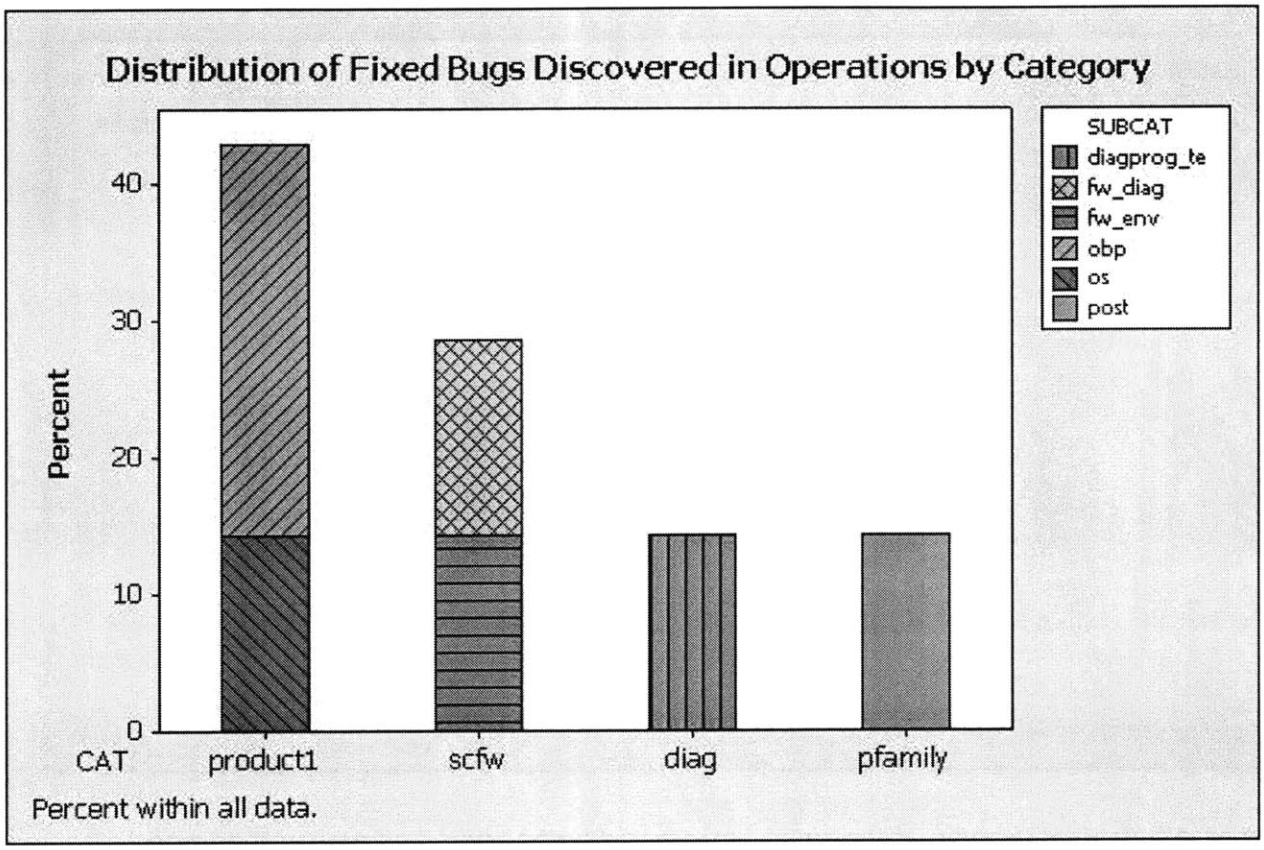

Figure 4.2.4: Distribution of bugs found in Operations by category and subcategory

4.3 Customer Escalation Analysis

The bugs that caused Customer Escalations in the field did not follow the category trend of bugs found in the field. Most of the escalations (88%) were due to OS Kernel bugs. The only non-kernel bug to cause an escalation in this dataset was an I/O (Ethernet) code issue for an add-on card. Figure 4.2.4 shows the distribution of escalated bugs by category. A very small percentage of bugs in this case study dataset were escalated by external customers.

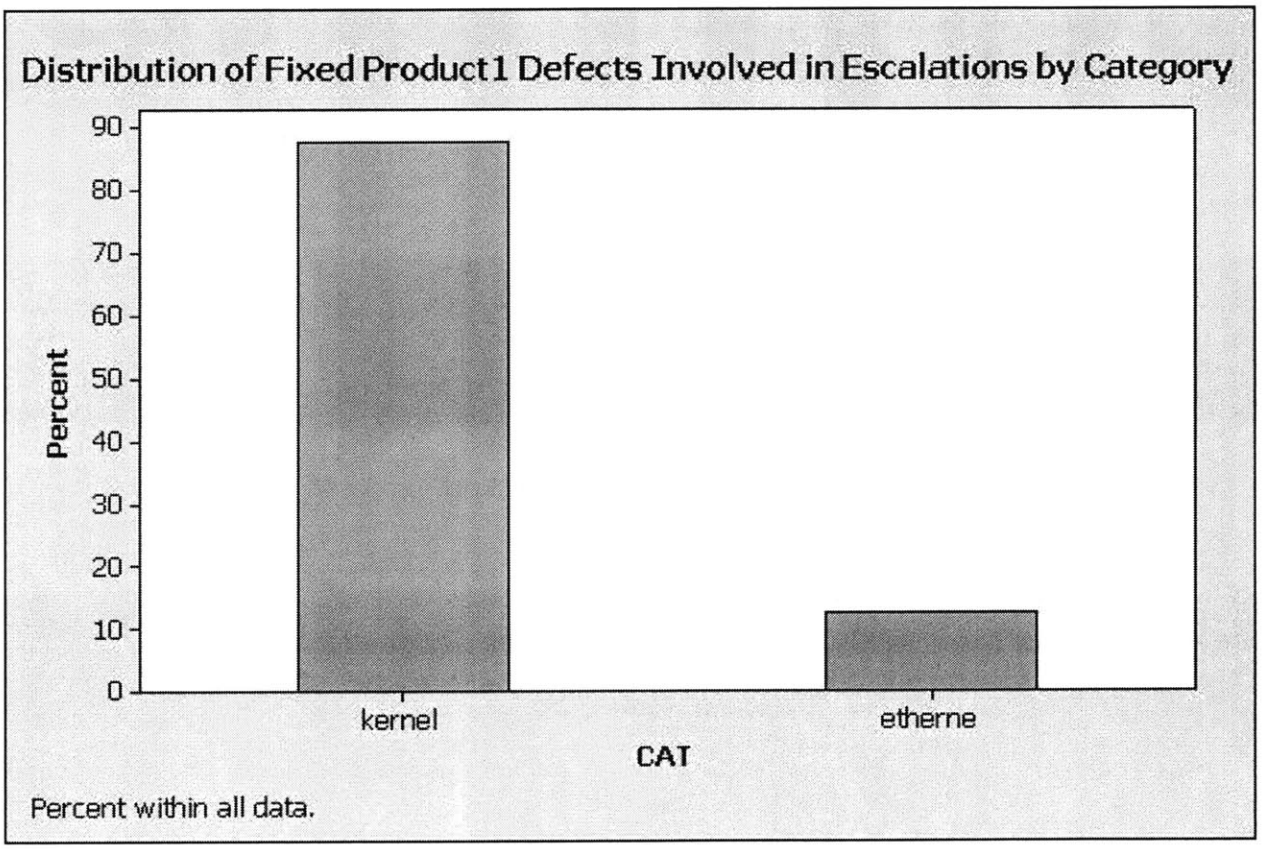

Figure 4.3.1: Distribution of escalated bugs by category

Kernel bugs and Ethernet bugs were also the categories most often assigned a high priority by the submitter. Figure 4.3.2 shows that priority 1 or 2 was assigned to 56 percent of kernel bugs and 60% of Ethernet bugs, compared to 30% or less for other categories.

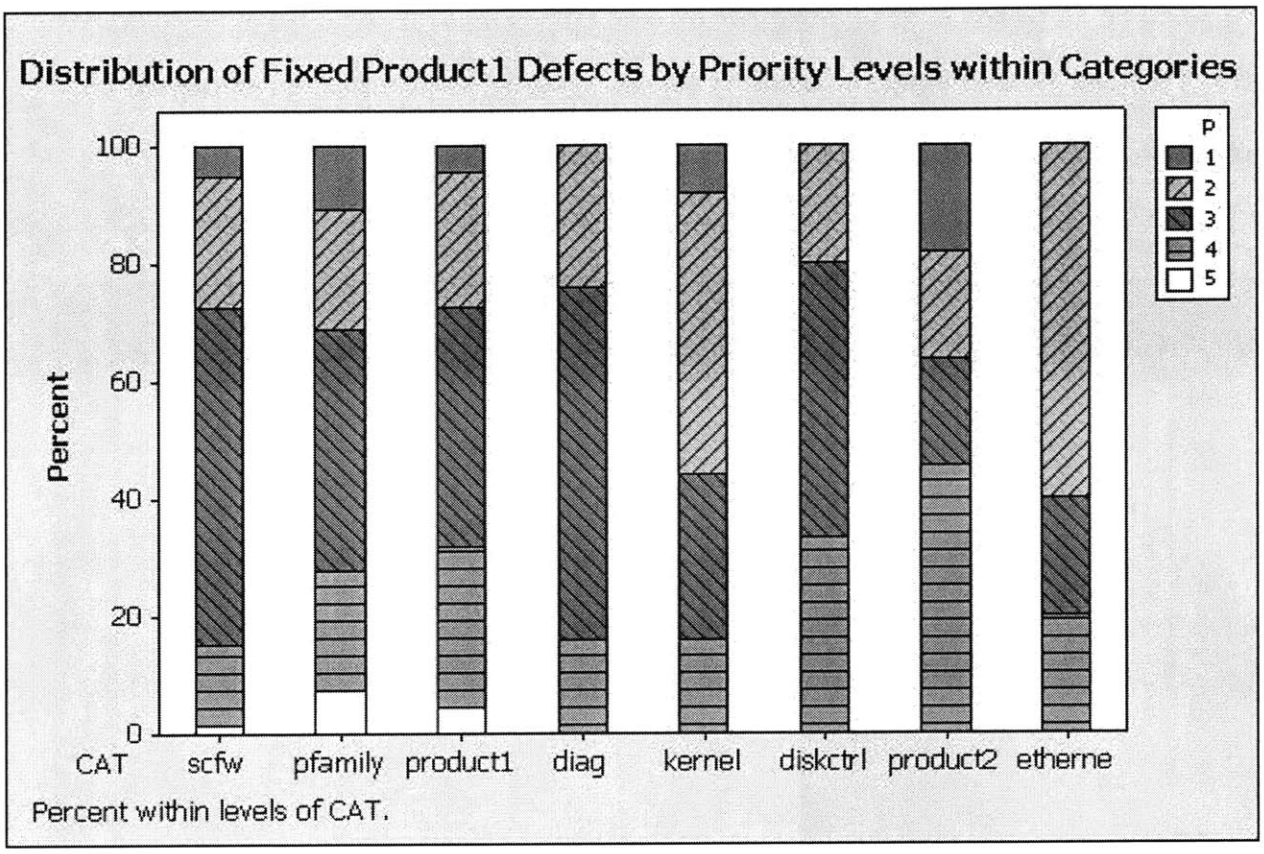

Figure 4.3.2: Percentages within the category of priorities assigned to fixed bugs

OS Kernel bugs may tend to cause customer escalations in part because they take longer to fix than other categories of software bugs. There seems to be a "threshold" behavior for some types of kernel bugs which are in production for a relatively long time before the fix is available. See figure 4.3.3.

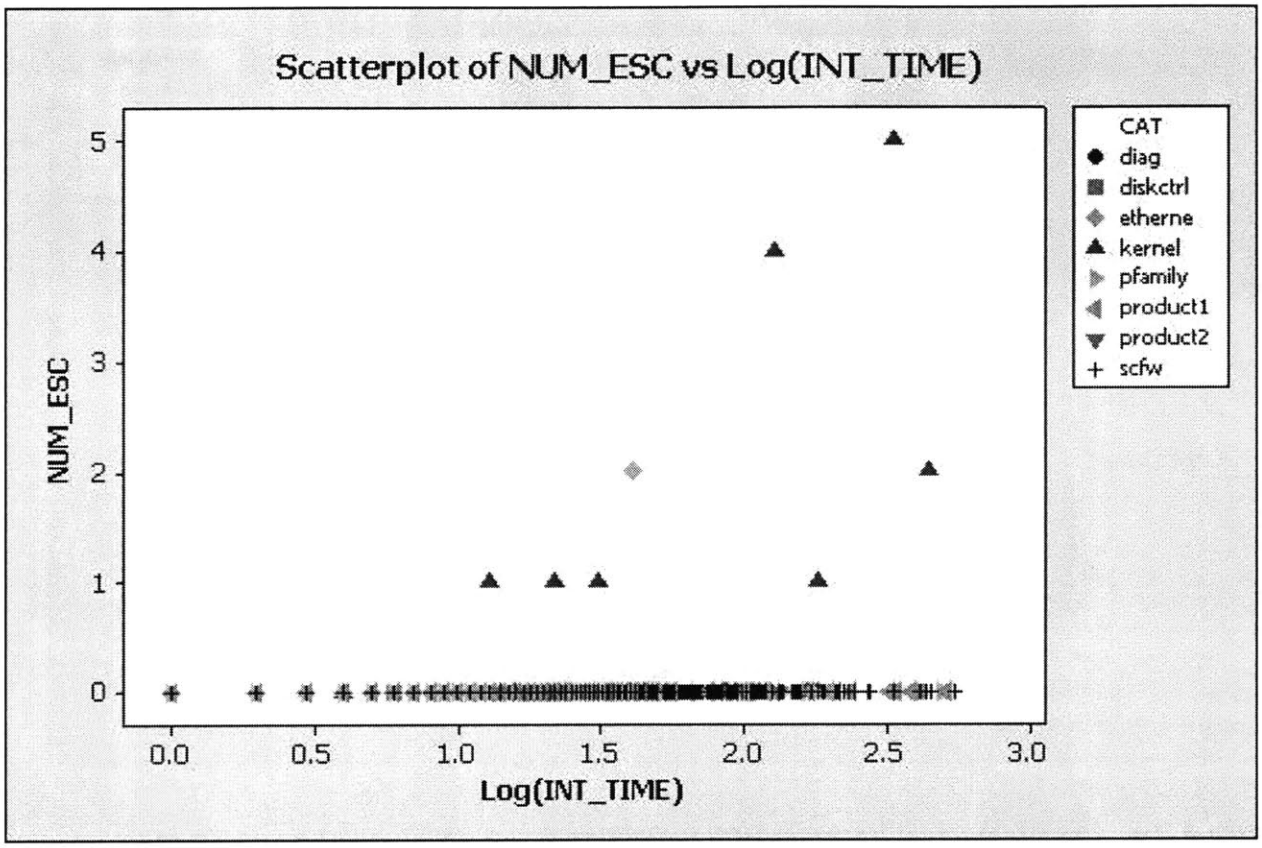

Figure 4.3.3: Scatter plot of Number of Customer Escalations vs. Log (Time to Fix + time to Integrate Fix)

More reasons for the prevalence of OS kernel bugs in escalations are discussed in Section 5. Bugs causing escalations were often discovered long before the escalation was initiated.

The users who discovered bugs that eventually caused escalations were distributed among Development, Field, Functional Test and System Test categories. Causes for Escalation-inducing

bugs were related to requirements in 50% of the escalation-causing bugs (either an incomplete or missing requirement). The other bugs that caused escalations can be traced to Coding Logic errors or conflicting actions among different modules of code. See Figure 4.3.4.

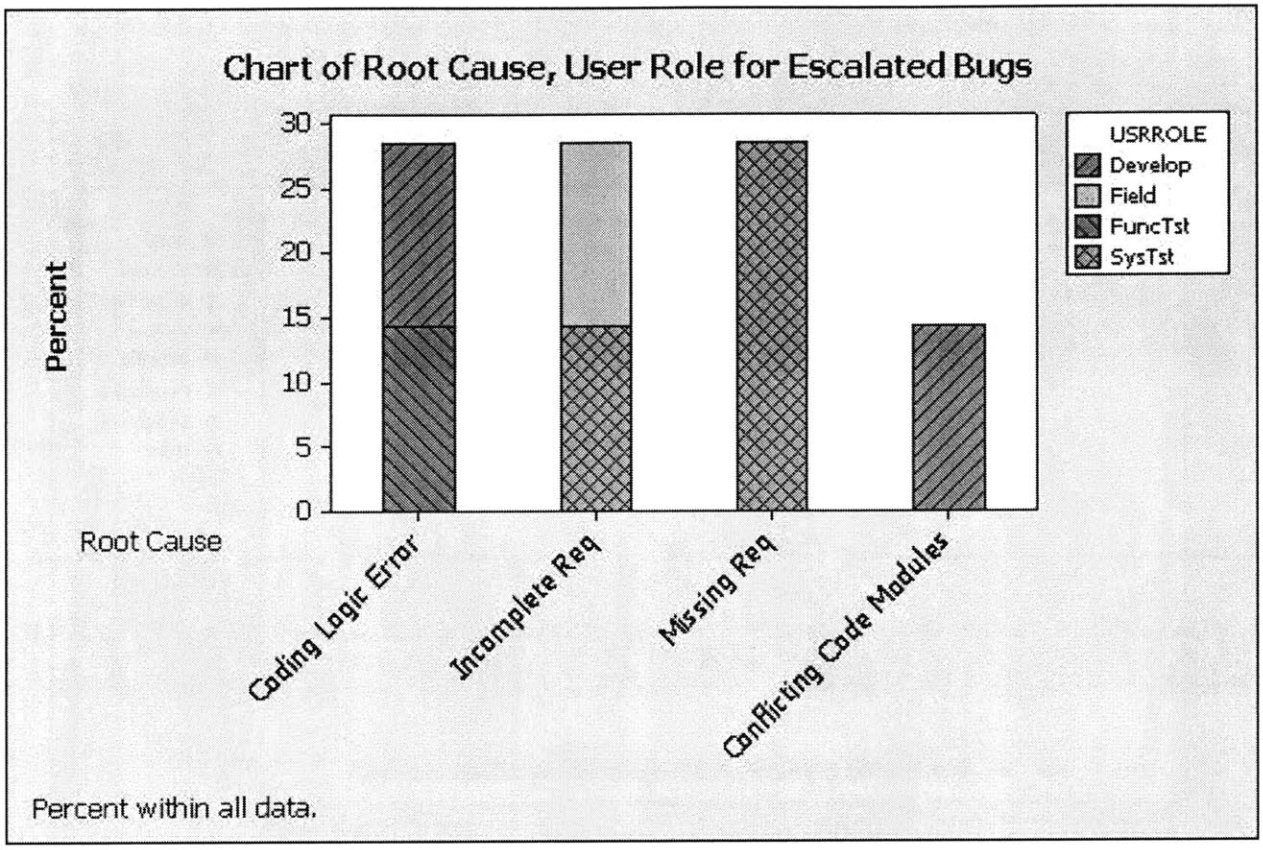

Figure 4.3.4: Distribution of bugs causing escalations grouped by Root Cause and Submitter Role.

Requirements problems (missing, incomplete, wrong or unexecuted requirements) as well as logic errors and code module conflicts also caused bugs that were found by the Field or Operations after being missed for the most part by developers and the code testing process. See Figure 4.3.5.

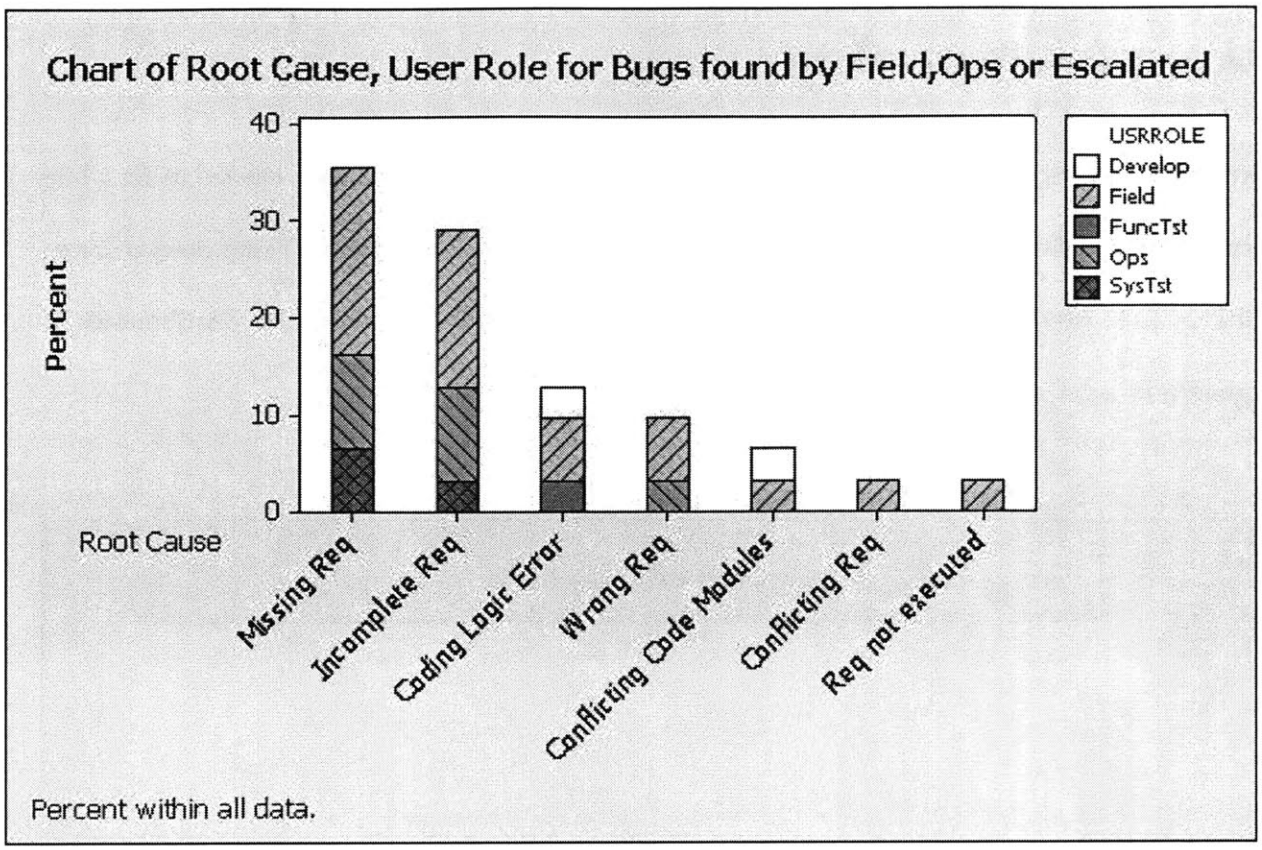

Figure 4.3.5: Distribution of Bugs escaping development test by Root Cause and Submitter Role

In this figure (4.3.5) I use the categories of missing, wrong or incomplete requirements to describe code that executes as intended by the developer, but has functionality that the bug submitter did not think was proper. The bugs categorized as "coding logic error" or "conflicting code modules" do not execute as intended by the developer due to logic problems or a code interface problem.

4.4 Integration Time Analysis

This database of bugs exhibited an overall Log-Normal distribution for the time needed to fix a bug and integrate the fix into the code base. The histogram of Log_{10} (Integration Time) does seem to exhibit at least two overlapping distributions. Figures 4.4.1 and 4.4.2 show the distributions of integration time on a linear and log_{10} scale respectively.

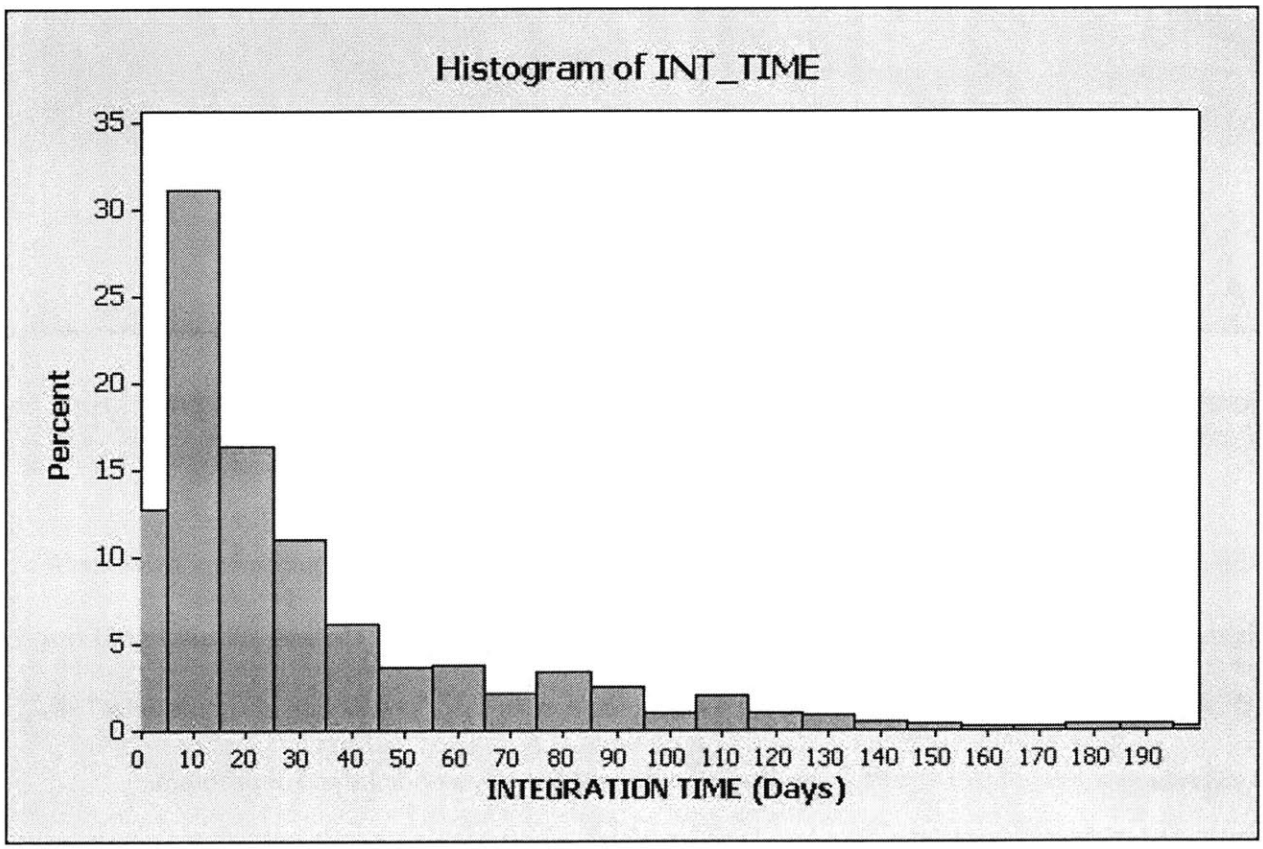

Figure 4.4.1: Histogram of time needed to fix and integrate defects

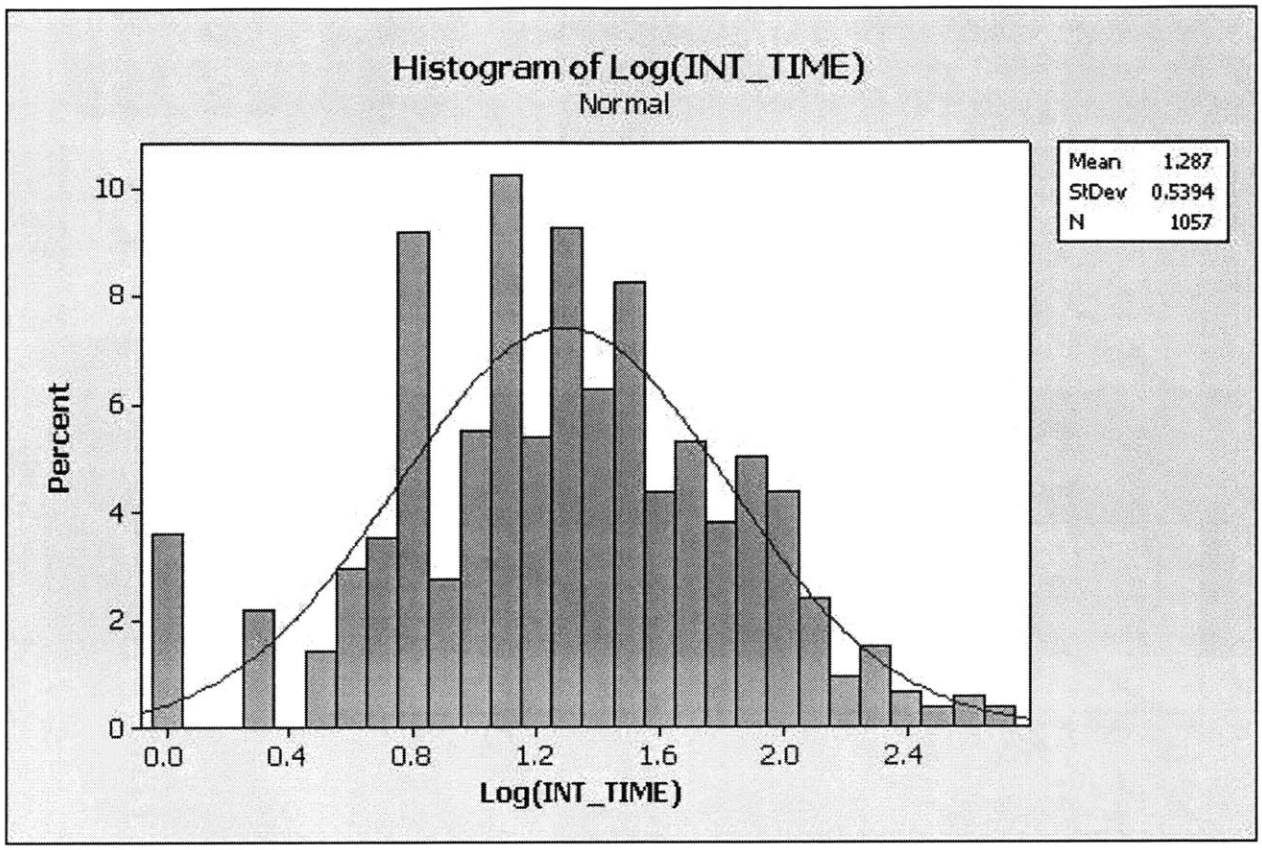

Figure 4.4.2: Histogram of Log_{10} of time to fix and integrate defects

The lognormal distribution was the best overall fit to this data, excluding the points pegged at an integration time of zero, which are artifacts of late bug submission. Figure 4.4.3 shows the cumulative distribution graph for integration time plotted on several scales. With a correlation coefficient of 0.996, the lognormal distribution was a clear choice.

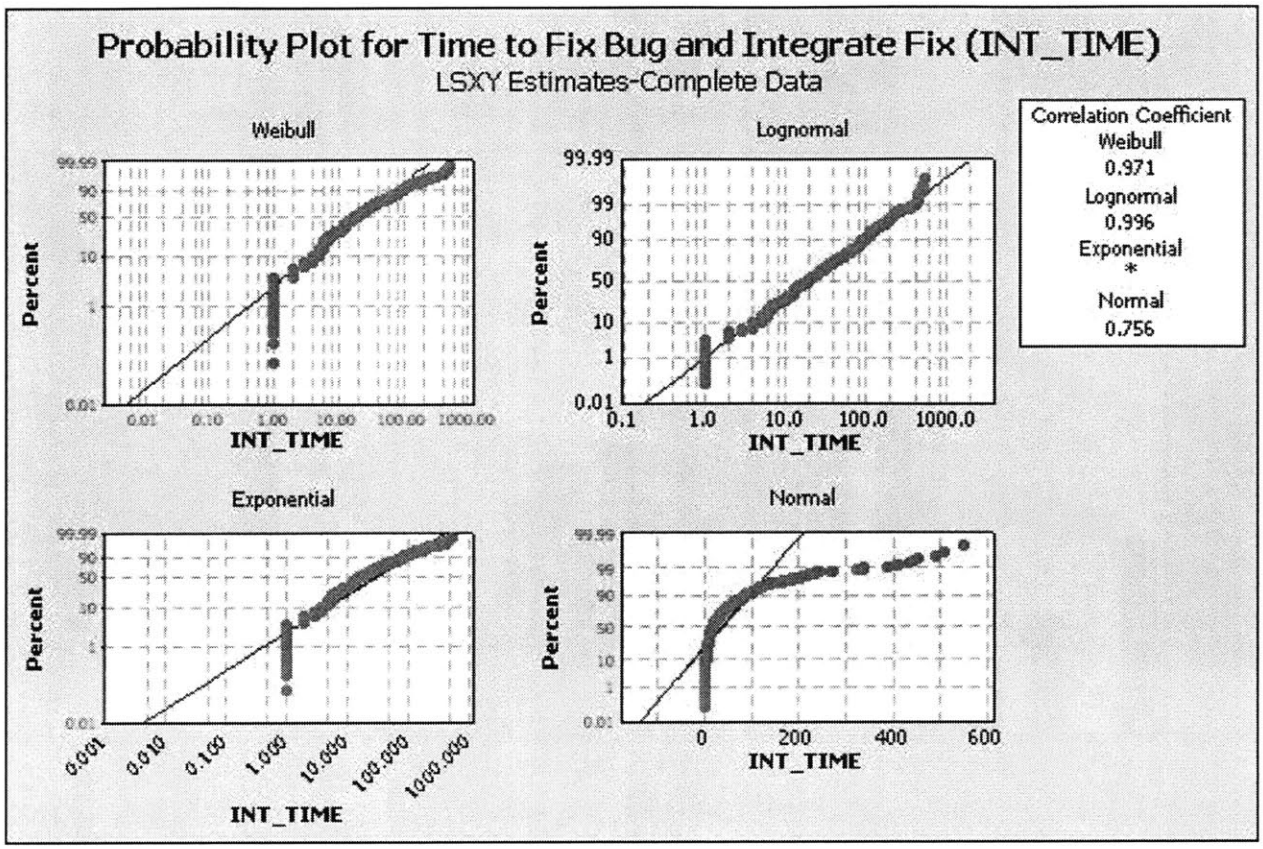

Figure 4.4.3: Cumulative distribution for Integration Time compared to modeled (straight line) distributions

Figure 4.4.4 shows box plots for Log_{10} of integration time for the categories of bugs in this sample. The OS Kernel code in particular has a longer integration time at the center of its log-normal distribution. (This is due in part to the fixed OS code build schedule- see commentary in section 5). The Ethernet code also has a distinctly long integration time. Firmware, especially P-family platform firmware, tended to have shorter integration times on average, but a very wide distribution in integration time.

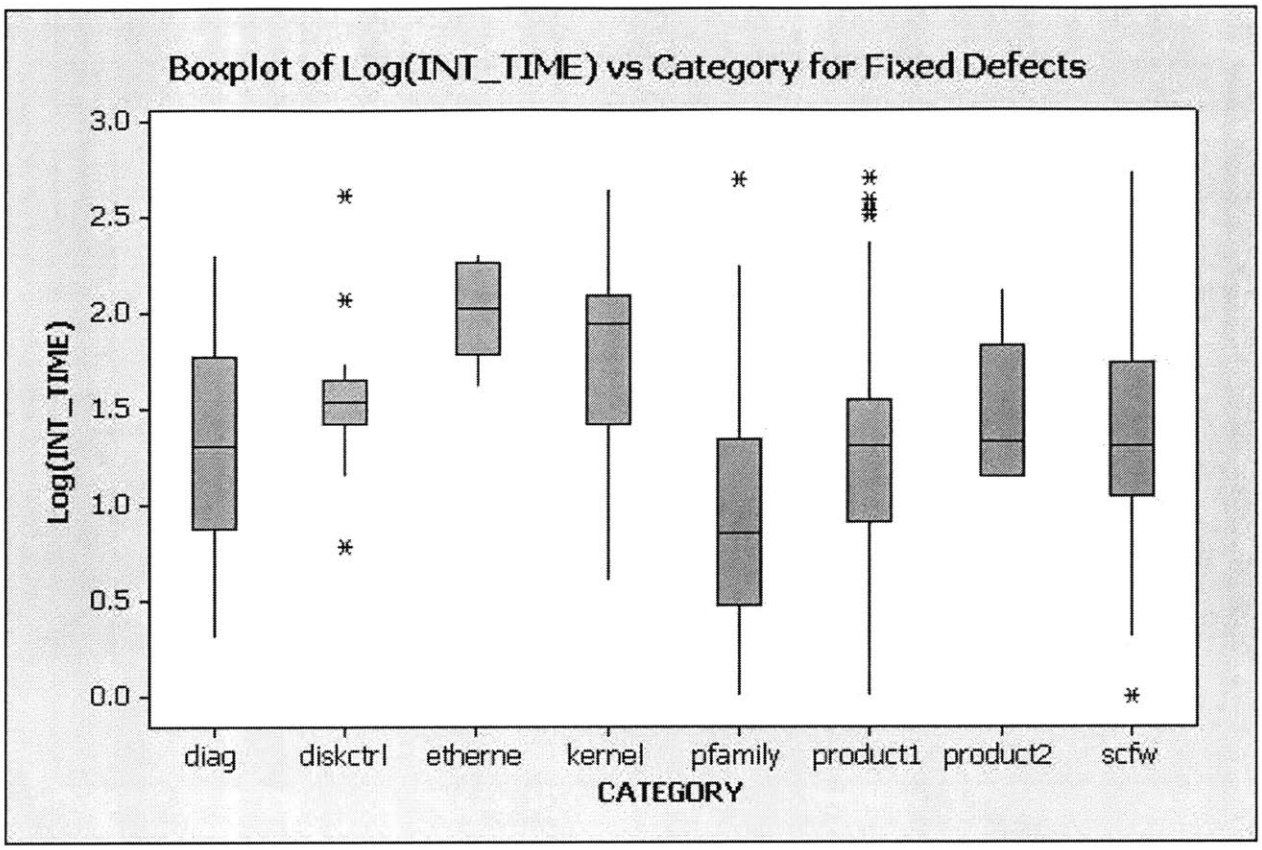

Figure 4.4.4: Box plots of Log_{10} of Integration Time for Fixed Bugs in Dataset by Category

The priority assigned to the bug by the submitter did not correlate well with the median integration time. The only effect a high (1) priority seemed to have on the distribution of integration times is that it reduced the length of the tail of the distribution on the high side. See Figure 4.4.5.

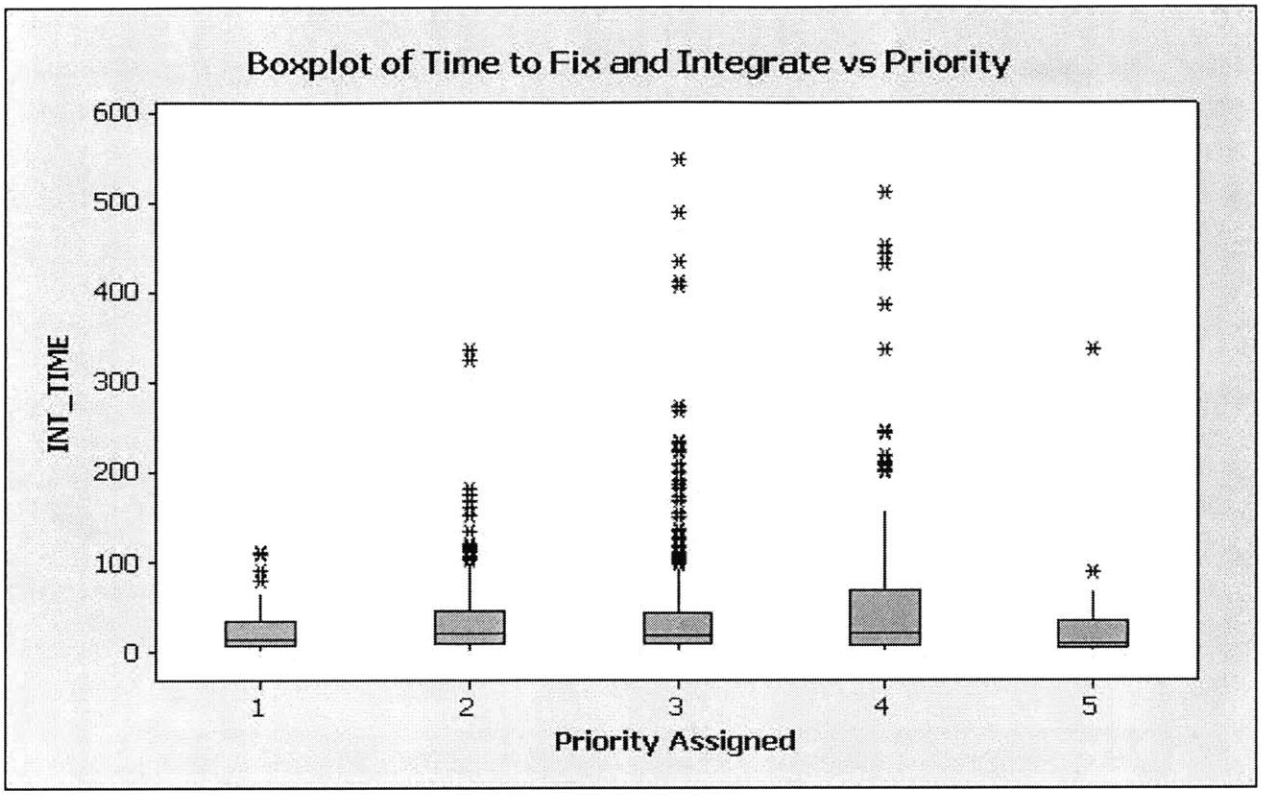

Figure 4.4.5: Time to fix and integrate bug vs. priority assigned to bug

Most of the differentiation in integration time among bugs could be accounted for by the main bug category. Figures 4.4.6 and 4.4.7 graphically illustrate an Analysis of Means on the normalized variable, Log_{10} (Integration Time) vs. the Priority assigned to the bug by the submitter for all bugs and for OS Kernel bugs only. The grand mean across the analysis categories is shown as the central line. The stepped lines around the center show the 95% confidence limit (given the sample size provided in the analysis category) that the mean of the analysis category is the same as the grand mean statistically. Points outside these limits are different than the grand mean with a P-value of 0.05. In figures 4.4.6 and 4.4.7, no priority level can be shown to have a statistically significant difference in mean integration time than the grand mean to 95% level of confidence. (If an 80% confidence limit was used instead, then a priority of 1 would have some effect on shortening mean fix and integration time).

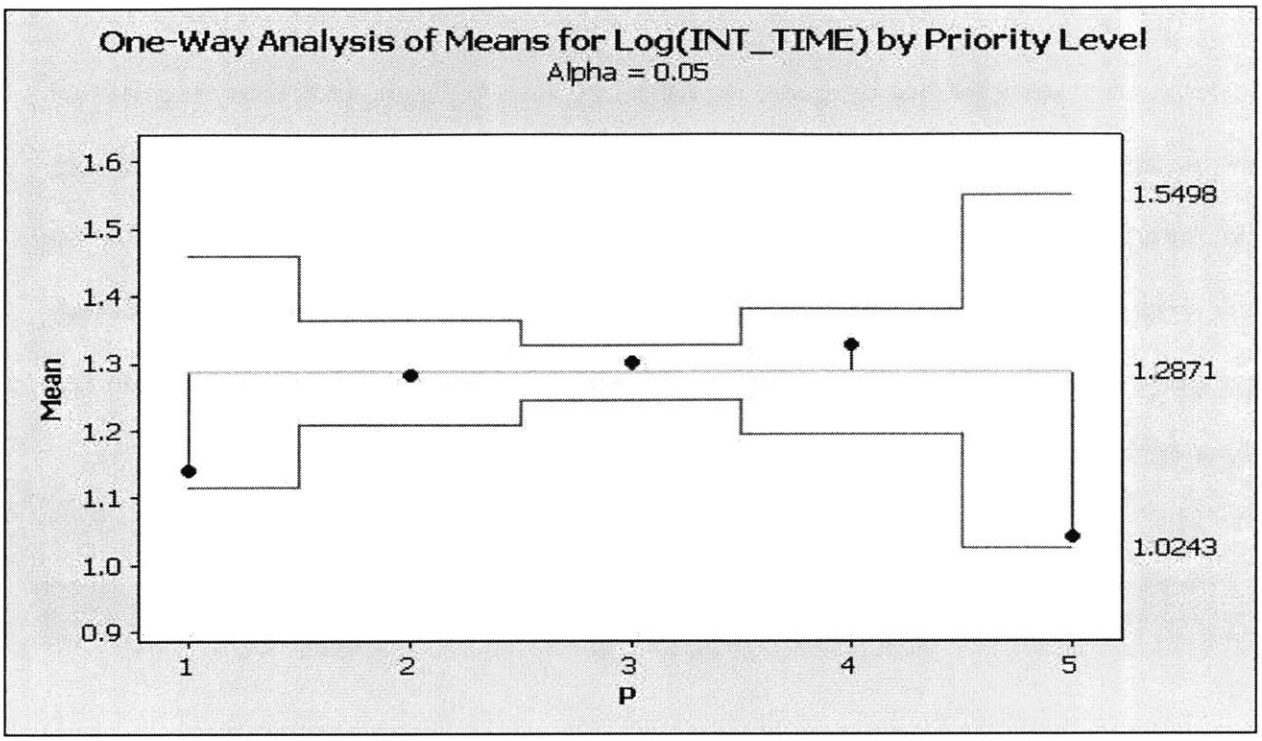

Figure 4.4.6: Analysis of Means Graph for Log10 Integration Time grouped by Priority Level

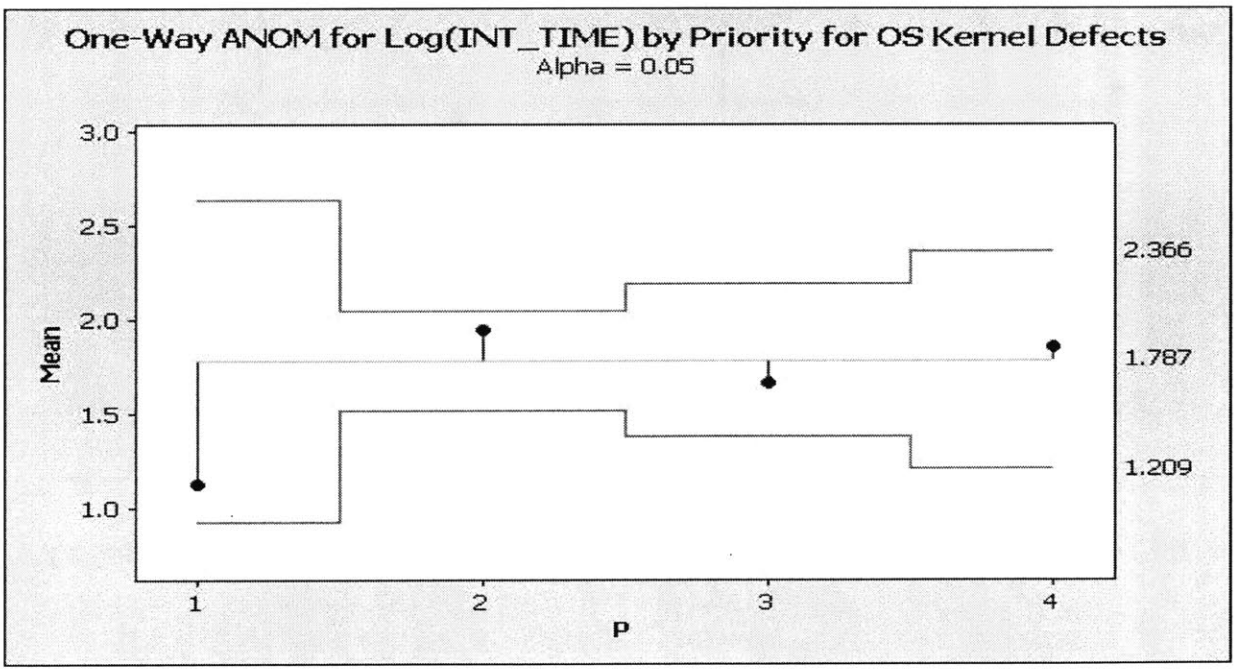

Figure 4.4.7: Analysis of Means Graph for Log10 Integration Time grouped by Priority Level for OS Kernel Bugs only

. However, when using the bug categories as analysis variables in Figure 4.4.8, four categories are shown to have a statistically different integration time than the grand mean. These are the P-family, SCFW, kernel and Ethernet categories. The SCFW category mean Log (Integration time) is just slightly above the grand mean, while the OS kernel and Ethernet firmware bugs have significantly longer integration times. P-family firmware, on the other hand has a significantly lower mean Log (Integration time) than the grand mean for all categories.

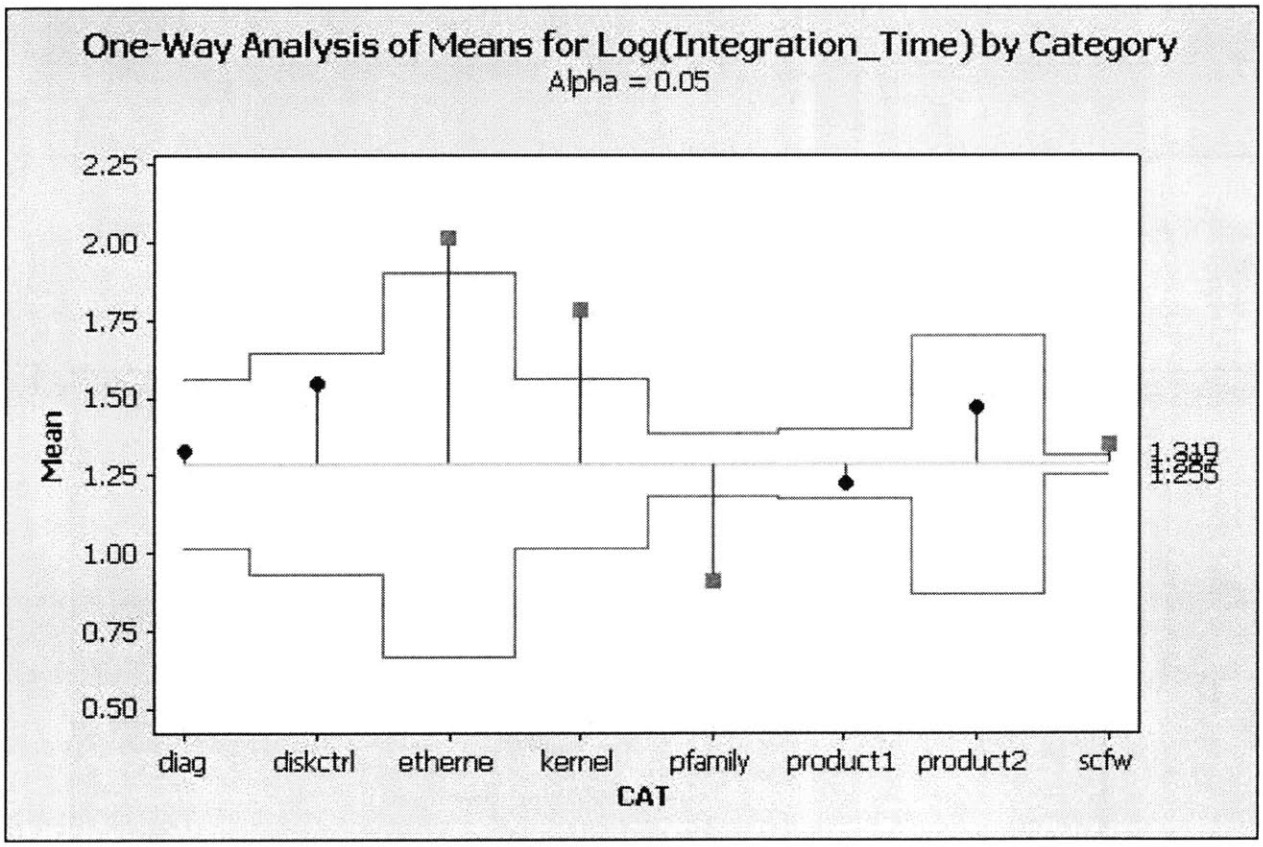

Figure 4.4.8: Analysis of Means Graph for Log10 Integration Time grouped by Category

4.5 What Groups found bugs

Bugs found were divided up fairly equally between Functional Test, System Test and Developers. Only about 2 percent of bugs were submitted by Customers or Field Service, and only about 0.68% of bugs in this data selection indicated in the database that they were found by Operations testing. Some of the bugs noted by Operations or the Field had been previously found by Development or Test, but were not fixed in time to prevent impact on these other groups. Figure 4.5.1 shows the breakdown by user (submitter) role.

The bugs discovered in the field and those found by Operations have an amplified impact due to the fact that they can cause customer dissatisfaction (for Field bugs) and interruption to prototyping schedules in the case of Operations bugs.

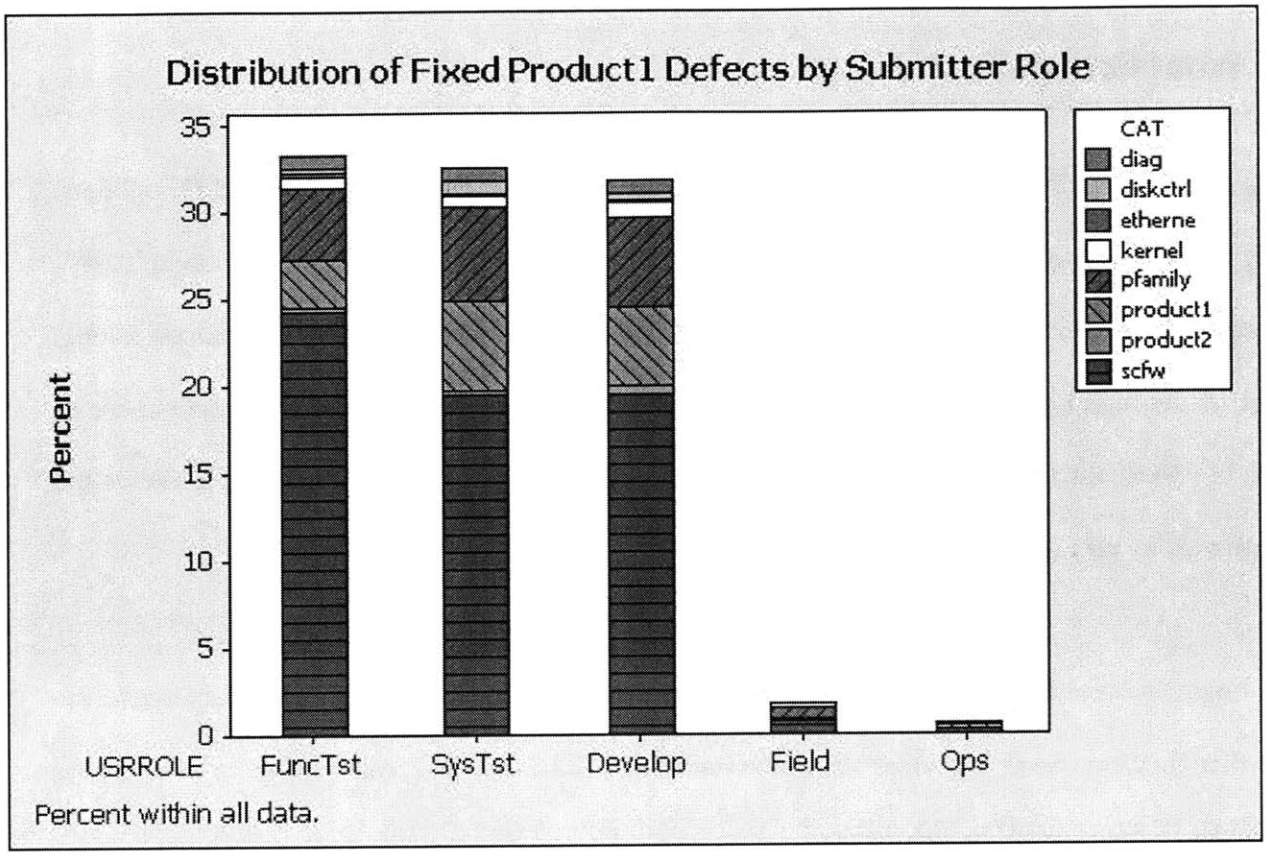

Figure 4.5.1: Distribution of fixed bugs grouped by submitter role

In the next section, possible explanations and implications of these results are discussed.

5.0 DISCUSSION

5.1 When Bugs were found during New Product Introduction

The distribution of bug record submissions for system software across the Product1 prototype builds was a success-factor for the program. The P0.2, P1.0 and P1.1 builds worked to expose the majority of system software bugs and absorb the majority of change requests. One might ask why the earliest builds, P0 and P0.1, did not also expose a similar number of defects and change requests. One reason is that P0 and P0.1 for Product1 were only tested by the development community, as planned. Developers do not typically start using the formal bug tracking system until they have a build of code that they consider complete and stable enough to do some integrated testing on their particular module. For P0 and P0.1, most of the bug records filed were cross-platform bugs found earlier on other P-family products to which the Product1 developers added a "customer call" as a courtesy to show replication of the bug on another platform.

One notable anomaly for the distribution of defects by prototype milestone is that a greater percentage of operating system kernel bugs associated with Product1 in this sample were found after GA than on most of the prototype builds. Kernel bugs that are specific to certain platform families, while rare, seem to be particularly difficult to expose during internal prototype testing. This result supports the importance of the efforts of large system software developers to replicate customer environments as closely as possible during part of their quality assurance testing[19].

5.2 Category/Sub-Category Analysis

5.2.1 The SCFW Category

There are several contributing factors to the dominance of the SCFW (System Controller Firmware) category for bug records submitted during Product1 new product development. I spoke to the manager of the SCFW effort during this time, one of the lead developers, and a QA test engineer who worked with Product1, to get more insight. Their comments can be paraphrased as follows:

- **The SWFW code was largely new.** The system controller firmware was the largest new software project for the P-family of products. . Although the code base size of other modules (such as the OS kernel) was much larger than that of the SCFW, the SCFW project had more new code targeted to the P-family products, whereas code such as the OS and the boot firmware had a large percentages of re-used code. Also, the resources available for the P-family efforts on the OS and boot code were greater than those available for the SCFW project. There were only six developers involved for the SCFW project. They were geographically dispersed, so global coordination was involved.

- **Requirements imposed on the SCFW code were evolving.** The architecture for the SCFW code was being developed in parallel with P-family hardware development. The SCFW architecture was more ambitious than previous generations of system controller firmware for this type of midrange server. The changes in the architecture included new interfaces to the host hardware and new functions on the system controller portion of the

platform hardware. Due to the parallel design and development of hardware and software, some of the design details for the software were not fleshed out until prototype hardware was in hand. This resulted in a series of changing requirements on the code during the hardware prototyping stages. The study by Javed et al[20] found a significant correlation between changing requirements in the coding stages and overall defect density.

- **Time pressure was a factor.** The program schedules for the initial P-family products such as Product1 and Product2 were very aggressive. The aggressive schedule goals compressed the time for software design additionally. (Lezak et al[21] mentioned that time pressure was related to 40% of the defects in their study).

- **The integration of older code modules into the new software was a challenge.** There was some code re-use in the SCFW firmware from previous generations of system controller firmware employed on midrange servers. However, this leveraged code was taken from three different code-bases from three different development groups. I speculate that any advantages of code re-use for this module were cancelled out in part by the disadvantages of integrating routines from three different groups of developers.

- **Hardware behavior needed to be accommodated by the SCFW code.** Because the system controller is the monitoring and management interface for the platform, it will tend to capture and report any hardware faults. However the definition of a "hardware fault" was evolving at the time of the prototype builds. There was a bit of a tug of war between what the software developers considered to be a fault and what the hardware developers considered to be a fault. At root, this could also be considered an issue of misunderstood

or evolving requirements with respect to what the system controller is supposed to report. Usually the hardware developers prevailed in these discussions as long as the "fault" was not critical to performance or functionality. Accommodation of hardware behavior resulted in a lot of SCFW changes and fixes to work around behavior that had not been anticipated in advance or was due to minor hardware faults.

- **The SCFW developers were conscientious users of BugDB.** The culture of the SCFW group encouraged software issue tracking and follow-up. This worked to their advantage in the long term.

5.2.2 Bugs which escaped software testing

The Pareto of bugs discovered externally reverse the order of the top two categories for bug counts and boost the OS kernel to position three. I believe the greater exposure of the p-family boot firmware and SCFW code compared to the Product1-specific code may lead to more external discovery of bugs. The kernel code in particular has huge field exposure in terms of number of systems, so its external bug count is really quite low in terms of defect density. The bugs reported in the P-family and SCFW firmware externally typically concern items visible to the user, and were not usually of high severity. (They were mostly severity 4 and 5). For example, they involved the wording of error messages and environmental messages. When Operations (hardware manufacturing) uncovered software defects, these defects were often part of the code modules that involved diagnostic code or were specific to the hardware product under test.

5.3 Customer Escalation discussion

Figure 4.3.2 illustrates that coding logic errors, conflicting code modules (interface problems) and requirements problems contributed to bugs escalated by external customers. These types of problems are commonly mentioned in the software engineering literature[22,23,24] although the categorization schemes and wordings for defect classification vary greatly. (For example "functional" problems and "requirements" problems are often the same types of defect looked at from different perspectives).

The few defects associated with customer escalations were mainly categorized as priority 2, with severity ranging from 1 to 3. (Priority 1 bugs are not shipped in production code, while priority 2 bugs may be permitted to ship under exception. Any priority 1 bugs involved in escalations were discovered in the field.) These defects were OS kernel bugs, with the exception of one bug filed on add-on card firmware.

Kernel bugs may be subject to more customer complaints for the following reasons:

- As mentioned in section 4, they take longer to fix and integrate. I'll discuss the reasons for this in the section on integration time analysis. A longer time to fix increases the exposure of the failure to the customer environment and also increases the number of customers who may exercise systems with the faulty software installed. Since these field failure modes are mainly intermittent and difficult to reproduce, they are harder to evaluate and debug. In addition, even if the fix is already integrated into an official OS build, the end customer will

not be able to get that fix until the build is release or a patch with the fix is released. Note that at the end of 2003, Sun accelerated its release of Kernel-related bug fixes.[25]

- In terms of a server's uptime, the kernel code will be in use more than most other pieces of software on the system. For example, the boot firmware is only executed during boot-up, and the system controller firmware may only be executing during system monitoring. However, the OS kernel needs to be active under all customer applications.

- The OS kernel code is used by a very large variety of hardware. This wide platform support makes for a large customer base.

- The OS kernel code is a very large group of modules in terms of total lines of code. Even with a very low defect density, there are likely to be more potential defects over a given timeframe in the OS kernel than in the other parts of the basic system software.

The above reasoning may also impact the priority and severity ratings given to kernel bugs by developers and field support. The tendency for escalations to occur on OS kernel code is a logical result of its size and deployment.

5.4 Integration Time Analysis

5.4.1. The Lognormal Distribution

Integration time was chosen as a variable for analysis, rather than the time to find a defect fix or the time to close out the bug record, because I believed it more accurately reflected the time from the

submission of the bug to the time the fix was generally available, either as a production release or as a patch.

The lognormal shape of the integration time distribution is an interesting result that has an intuitive explanation. Lognormal distributions are common for many natural and social phenomena.[26,27] Some examples of lognormal distributions from the previous references are:

- Mineral resources in the earth's crust
- Survival times after diagnosis of cancer
- The age of marriage in human populations
- Economic data such as the size of corporations

Lognormal distributions can be modeled in a similar way to normal distributions, as a result of many small random effects. However, in the case of lognormal distributions, the small random effects combine in a *multiplicative* manner rather than in an additive manner. Thus, the resulting distribution is skewed to the right as a result of the multiplicative central limit theorem[28]. Joe Marasco, Former VP of Rational Software, suggests the lognormal distribution as an appropriate starting point to model the probability of success of software projects.[29]

In some ways, it is encouraging that the time-to-integration distribution appears lognormal. Although it implies that several small issues may combine multiplicatively to create a very long time to fix, it also implies that we can improve our success in fixing defects on a log scale rather than a linear scale by attacking only a few critical factors.

5.4.2 Integration Time Category Analysis

The most noticeable feature of the integration time distributions for different software categories is that the OS kernel and Ethernet firmware category distributions have significantly longer integration times. I will speak to the longer integration times for the OS kernel bug fixes first.

1) The OS is among the large pieces of software at Sun required to follow a very structured development[30] processes. This involves many approval stages that the smaller firmware modules do not require in their development processes.

2) OS builds and patches go through a more extensive System Test and Performance Regression Test[31] cycle than firmware, which is much smaller in total lines of code, and limited in platform scope. The performance and regression testing occurs in parallel with the fix verification testing. A wide range of SPARC™ and x86 architecture servers are used in the System Test cycle, and multiple customer scenarios are stimulated. I counted 56 separate test suites run for patch testing in the SunSolve™[32] update referenced. The SunSolve™ article was up to date as of January 2003. Additional test time is consumed if the bug fix needs to be verified in more than one released version of the OS.

3) The types of bugs which that cause escalations are typically difficult to reproduce and debug. Escalated bugs caused by coding logic errors or conflicting code modules tend to be particularly elusive.

To get more insight into the reasons behind why it takes more time to fix and integrate these types of defects and why these particular defects caused escalations, the text comments and evaluations for the kernel bug records with more than one escalation filed against them were examined individually. The sample size for this examination was quite small, so my comments need to be taken with that caveat in mind. I've summarized a few interesting cases below:

In one case the long time to fix was due to an initially low priority (4) set on the bug. The bug involved an error message that was considered functionally benign. In addition it was very difficult to reproduce. So, the bug record sat at status "accepted" for an extended period of time due to other priorities. However, customer perception of benign error can be different than that of a developer, especially a customer who closely monitors the message files. Once efforts were restarted after the customer complaints, the bug fix took only about a month to complete.

In another case, the priority was set at 3, and there actually was a bug fix available before the customer escalations were filed. (The reason the priority was set at 3 rather than higher was that a user would have to execute what seemed to be an uncommon series of actions to invoke the problem.) The escalations were initiated because the bug fix was not made available in released build or patch form until several months after the fixed code was available. Uncovering the reasons for the delay in integrating the bug fix into a patch is out of the scope of my study. However I can guess that the medium priority initially set on the bug was part of the problem, because the patch for this bug was queued behind that of another bug mentioned in the escalation report. The accelerated release process for OS patches[33] initiated at the end of 2003 should ameliorate problems of this sort.

A third interesting case was filed at a high priority (2). It was a difficult coding logic bug, and there was some discussion between developers as to the validity of the original root-cause theory. The bug was inactive for several months between the initial evaluation by one developer and the submission of more comments by another developer. Examination of an escalation report showed that the bug proved very difficult to reproduce at the customer site. A temporary binary was given to the customer, who then stopped complaining about the problem. The customer delayed a long time in testing the new binary, and also had problems reproducing the problem. When the new binaries, which had finally been tested and accepted at the customer site, were sent to the code inspection team for internal Sun review, more questions were raised about the validity of the fix and whether it really addressed the root cause, which was assumed to be a rare race condition. Another patch was produced in parallel with the start of the process to integrate the fix into production code, and it took another several weeks to discuss the concerns and then to address the concerns of the code inspection team.

Another delay was incurred at the release engineering level. This seems to be the point at which the second developer entered his comments and effected some changes to the fix. There was then more delay for testing necessary to start the process of integrating the fix into production code and for production of the official patch, which had the second developer's changes.

In this small sample of OS kernel bug escalations, a number of factors, which may be multiplicative rather than additive, contributed to the long time between bug submission and integration of the bug fix into production code:

- There can be problems with initial prioritization of the bug. A priority lower than 1 or 2 may put a bug fix behind other work in terms of developer evaluation, code inspection, regression testing and release.
- Many times there are problems replicating the defect. Difficulties with replication slow up both evaluation and verification.
- Communication delays can slow down bug fixes. Communication delays between customer and developer consume time when a customer environment is needed for verification of a fix.
- Technical differences between interested parties about the root cause of the bug can slow down evaluation.
- The OS review and testing process for both patches and integration into new builds is thorough, as it should be. The fixed cycle time due to the verification and test process cannot be further compressed without additional resources. Reducing this delay would require a cost/benefit trade off on the part of Sun.

Section 4 also examined the distributions of integration time for various bug priority levels. No statistically significant relationship could be found between assigned priority level and integration time, with the exception of a marginally significant effect on shortening integration time for a priority level of 1. A check of the relation of priority to integration time within categories showed no better correlation than that across categories. These results could be assigned to problems with the priority assignment process or problems with the assignment of resources to bug fixing, or both. Other factors than the priority assigned seem to be driving the fix rate of most of these bugs.

5.5 The Role of the Bug Submitter

Product1 had a successful software development cycle in that its team exposed most of its software faults during Development and Quality Assurance testing. The even distribution of bug submission between developers, functional test and system test could be interpreted as a good usage of the BugDB tool by different parts of the organization.

The submitter-role analysis deserves some caveats, however. The definition of developer, system test and functional test (which were allowed categories for the submitter role) was not always clear to the bug submitter. For example, hardware developers submitted some firmware bugs under the "developer" role even though they were not the developers of the code. Perhaps their submissions might be better classified as "system test". I also noticed that there was no submitter-role category available for operations testing. I therefore had to look at the submitter name or the interest list attached to find bugs that were discovered during manufacturing test. Another problem is that the Operations product engineer assigned to Product1 would often refer a suspected bug to a software developer rather than submit it himself, as a matter of courtesy. As a result, the percent of bugs discovered by Operations in this sample may be underestimated. This indicates that there are different thresholds for submission of a bug across the product team.

6.0 SUMMARY

The overall objectives of this thesis were to use the Product1 case study defect dataset to extract information that might lead to improvements in our firmware and software development processes. The final chapter will present some conclusions, recommendations and proposals for further work based on this research.

6.1 Conclusions

This research included data extraction of a dataset of firmware and system software defects from a large relational defect database. The data was limited to defects exhibited on a particular hardware platform during the new product development and initial production stages. Graphical and statistical analysis of the data was supplemented by informal interviews with key project personnel to supplement understanding of the results. The analysis can be summarized with the following findings:

The majority of defects in the dataset were discovered during the P0.2, P1.0 and P1.1 build phases. Discovery of defects during the early prototype builds is a desirable result. The OS kernel category was the only defect category that exhibited as large a percentage of faults after GA as during P0.2, P1.0 or P1.1. This finding supports the supposition that it is difficult to replicate the spectrum of customer environments during development phase testing of a code module as large as the OS kernel.

For this dataset, code reuse greatly reduced defect density during the development and testing stages compared to that of mostly new code based on a new architecture and new requirements. However, the new code performed relatively better than similarly sized modules of mostly re-used code later in the life of the project. The conscientious tracking of defects on the SCFW code module during development may have contributed to success in its maturity.

External customer complaints about released software exhibit a threshold behavior with respect to the system hours of exposure to defective code. Code with a smaller customer base and a shorter median time-to-fix does not seem to trigger escalations, even if a greater total number of bugs are discovered on it externally.

Bugs discovered external to the software development process were split between "requirements" (functional) problems and "coding logic" (execution) problems. For all stages of development there were more requirements-related root causes than coding-related root causes. For escalated bugs, the problems were evenly split between these two major categories.

The time to fix and integrate fixes for these software defects back into production code had a log normal distribution. The log normal distribution implies that the factors contributing to integration delays interact multiplicatively, rather than additively. As a result, the elimination of a few key delay factors could greatly reduce the time it takes to fix bugs.

Our present bug prioritization methods are not effective at reducing total time to fix or in predicting what bugs may become customer escalations. The only significant effect a high assigned priority has on the time to fix and integrate a defect is to reduce some of the outliers in the

fix-time distribution. Sun is mindful of this problem and is currently working on projects to address it.[34]

Communication problems delay progress on bug evaluation and verification. Communication delay applies to internal communication between functional groups and communication between the developers and external customers. The intermittency of many defects is another big factor that delays bug evaluation.

6.2 Recommendations

Many recommendations for improving defect tracking and software development have been suggested throughout the chapters of this thesis. I've summarized these recommendations, and added others in the sections below. The recommendations are divided into development process, escalation process and defect tracking categories. These recommendations are targeted specifically to Sun system software. However they may be applied, in principle, to other software development environments.

6.2.1 Software Development Process Recommendations

- The development process for system software needs to be tailored to the characteristics of the module under development before development actually begins. A project with a new architecture and a large percentage of new functionality needs more front-end attention in the architectural and requirements phase than does a module that is mostly reused. Modules with a lot of new code require more resources and more scrutiny during the early architecture and design phases of development. The time spent in architecture review and

cross-functional requirements reviews that results in a more stable set of requirements will be given back to the project and amplified due to better code quality.[35]

- It would be beneficial to investigate new methods to create more understandable and complete requirements for all new functionality. See Leveson's article on Intent Specifications.[36]

- Progress has already been made towards replicating the spectrum of customer environments during OS system testing. Further progress could be aided if more root-cause analysis was performed and accumulated on defects only exposed in customer environments.

6.2.2 Escalation Prevention Recommendations

Escalations are costly to support because they require intensive attention from the field service and sustaining software teams. Some recommendations for reducing the risks of escalations, based on the examination of escalation records in my thesis dataset, are listed below:

- A new project has already been initiated at Sun to improve bug prioritization methods. I would ask this team to look into doing more than just improving the priority field in BugDB. To effectively predict which bugs need rapid engagement, one needs to be able to model the cause and effect relationship between characteristics of the defect and the result in the field. One approach to modeling these cause and effect relationships would be to leverage the escalation prediction modeling already going on at Sun[37] and to use it as an input to the BugDB priority rating. An alternative approach would be to create a better description of bug characteristics through a set of fields (not just one priority number) and use this

comprehensive bug classification scheme for a variety of applications. The Orthogonal Defect Classification[38] standard is one defect classification scheme that has been widely applied to code quality improvement and directing software development resources toward the areas in most need them[39].

- The escalation process itself may be in need of some logistical improvement. The small sample of escalation details that I've looked at implicated some problems getting hold of the right customer contacts to communicate with about bug follow-through and fix verification. Insuring that the management at customer sites buys into customer participation in bug replication and fix verification up front would be helpful. Delay in reaching the customer for fix verification has a direct impact on the length of time other customers are exposed to the bug.

- Do developers involved in bug fixes have clear guidelines on when to ask for help when they are stuck in the evaluation phase? If some clear internal escalation processes for bugs that met certain criteria (category, priority, time since submitted) were established, bugs would be less likely to languish in the evaluation phase.

6.2.3 Bug Database and Defect Tracking Recommendations

- The first step in applying institutional learning to the data stored in BugDB could be as simple as making the "Root Cause" field in that database mandatory in order to move a bug record to "fixed" status. If properly designed, the "Root Cause" field would be a good pointer to areas in need of software development process improvement.

- There was no Operations Test "user_role" category available in BugDB. Similarly, the "Developer" entry for the "user_role" category does not differentiate between hardware developers and software developers. The database designers should update the designations permitted for bug submitters and customer contacts.

- The software group leaders who define BugDB categories and subcategories for a specific program should avoid using "other" in the name of a category or subcategory where possible. If a category or subcategory with "other" in its name is collecting a significant number of bug records, it deserves to be subdivided into different categories with meaningful names.

- In order to draw accurate conclusions from data in the defect database, those who input data need to keep it as accurate as possible. For example, developers should submit bugs when they discover them, not when they fix them. Submitting bugs when discovered would not only permit more accurate data analysis in hindsight, but it would also allow other groups involved in platform development to recognize bugs as they occur and add customer calls to the database. Additional customer calls could help the developers reproduce and debug the problem.

- A standard should be developed so that developers of the software modules making up a program have uniform criteria for initiating use of the BugDB tracking tools. Use of the tool too early, when code is very unstable, slows down progress without adding value. On

the other hand, delaying use of the tool too long is also counterproductive because developers may lose track of early bugs which become more troublesome at a later stage.

- The BugDB easy reporting tool does not allow one to extract bugs based on the hardware platform on which the bug appears. Many hardware platform QA teams need to do exactly this kind of extract, so the addition of that capability would be helpful.

6.3 Suggestions for Further Work

The bug tracking database we use at Sun can be a rich source of institutional learning. A great deal of project history is stored there. In particular, a comprehensive Root Cause Analysis project based on BugDB data should be established. Initially, a Root Cause Analysis project could be based on manual review of bug records based on sampling[40]. A more automated and comprehensive Root Cause analysis project could be supported by additional fields in the BugDB database, such as the ODC[41] categorization fields previously mentioned. I feel confident that the ROI for such a project would be substantial.

7.0 APPENDICES

```
#!/bin/csh -f
/usr/dist/exe/isql -BugDB -Account -Password     <<_EOF
set nocount on
select distinct
'P' = convert(varchar(1),priority),
"|",
'S' = convert(varchar(1),severity),
"|",
'BUGID' = convert(varchar(7),bugid),
"|",
'CAT' = convert(varchar(7),cat_cat),
"|",
'SUBCAT' = convert(varchar(9),cat_sub),
"|",
'SBMTR' = convert(varchar(7),submit_operatr),
"|",
'SYNOPSIS' = convert(varchar(30),synopsis),
"|",
'BUG' = convert(varchar(3),bugrfe),
"|",
'SUBDATE' = convert(varchar(11),submit_date),
"|",
'INTDATE' = convert(varchar(11),integrate_date),
"|",
'DUPLIC' =  convert(varchar(7),duplicate_of)

from
        bug_tbl,
        bg_cal_t,
        pipe_tbl
where
        bugid_call = bugidue
        and
        bugid_pipe = bugidue
      and
        ((cat_cat = 'product1' and cat_sub like 'os')
      or
      (cat_cat = 'kernel'
                and
                (hardware_version like 'product1'
                or
                hardware_version like 'product1_server'))
      or
        (cat_cat = 'pfamily'
                and
                (hardware_version like 'pfamily'
                or
                hardware_version like 'product1_server'))
      or
        (cat_cat = 'ethernet_iochip'
                and
            hardware_version like 'product1_server')
      or
```

```
        (cat_cat = 'firmware'
                and
            hardware_version like 'product1_server')
      or
        (cat_cat = 'mpt'
                and
            hardware_version like 'product1_server')
      or
        (cat_cat = 'product2'
                and
            hardware_version like 'product1_server')
or
        (cat_cat = 'diag'
                and
            hardware_version like 'product1_server')
      or
      (cat_cat = 'scfw'
                and
                (hardware_version like 'product1_server'
                or
                hardware_version like 'product2_server')))
order by
      cat_cat,
      priority,
      severity,
      submit_date,
      bugid
go
_EOF
```

Figure A1: Example of SQL Query on BugDB

8.0 REFERENCES

[1]External developer access to BugDB is at http://bugs.sun.com/bugdatabase/index.jsp

[2]For Sybase™ enterprise database technical information see http://www.sybase.com/developer/products/asetechcorner

[3]For Oracle® enterprise database technical information see http://www.oracle.com/technology/documentation/database10g.html

[4]Tilmann Bruckhaus, Charles Ling, Nazim Madhavji, Shengli Sheng, Software Escalation Prediction with Data Mining, Proceedings 20th International Conference on Software Maintenance, 2004, Workshop on Predictive Software Models, see http://www.site.uottawa.ca/psm2004/PSM-2004-Program.html

[5] MSR 2004, Proceedings 1st International Workshop on Mining Software Repositories, http://msr.uwaterloo.ca/program.html

[6] ISCE 2004, Proceedings IEEE 2004 International Conference on Software Engineering

[7] Thomas J. Ostrand and Elain J. Weyuker, A Tool for Mining Defect-Tracking Systems to Predict Fault-Prone Files, International Workshop on MSR, W17S Workshop, 26th International Conference on Software Engineering, 2004, p85-9

[8] Lezak, M. Perry, D.E., Stoll, D., A Case Study in Root Cause Defect Analysis, Proceedings of the 2000 International Conference on Software Engineering, p. 428-437

[9] Parastoo Mohagheghi, Reidar Conradi, Ole Killi, Henrik Schwartz, An Empirical Study of Software Reuse vs. Defect-Density and Stability, Proceedings of the 26th International Conference on Software Engineering, 2004.

[10] Talha Javed, Manzil-e-Maqsood and Qaiser Durrani, A Study to Investigate the Impact of Requirements Instability on Software Defects, ACM Software Engineering Notes, May 2004 Volume 29, Number 4.

[11] Tilmann Bruckhaus, Charles Ling, Nazim Madhavji, Shengli Sheng, Software Escalation Prediction with Data Mining,

[12] Dewayne Perry, Harvy Siy, Lawrence Votta, Parallel Changes in Large-Scale Software Development: An Observational Case Study, ACM Transactions on Software Engineering and Methodology, Vol. 10, No. 3, July 2001, p.308-337.

[13] For ISQL tutorial see: http://download.sybase.com/pdfdocs/asp1200e/uxutil.pdf

[14] For a Solaris Korn shell reference see: http://docs-pdf.sun.com/816-5165/816-5165.pdf

[15] Lezak, M. Perry, D.E., Stoll, D., A Case Study in Root Cause Defect Analysis, Proceedings of the 2000 International Conference on Software Engineering, p. 428-437

[16] Jutta Kreyss, Michael White, Steve Selvaggio, Zach Aakharian, Text Mining for a Clear Picture of Defect Reports: A Praxis Report, Proceedings of the Third IEEE International Conference on Data Mining (ICDM'03)

[17] http://www.minitab.com

[18] A good t-test description is at http://www.itl.nist.gov/div898/handbook/eda/section3/eda353.htm

[19] Willa Ehrlich, John Stampfel, Jar Wu, Application of Software Reliability Modeling to Product Quality and Test Process, 12th International Conference on Software Engineering, Feb 1990, p.108-116

[20] Talha Javed, Manzil-e-Maqsood and Qaiser Durrani, A Study to Investigate the Impact of Requirements Instability on Software Defects, ibid

[21] Lezak, M. Perry, D.E., Stoll, D., A Case Study in Root Cause Defect Analysis, ibid.

[22] Lezak et al, ibid

[23] Purushotham Narayana, Software Defect Prevention – In a Nutshell, Six Sigma Software/IT, http://software.isixsigma.com/library/content/c030611a.asp

[24] Robin Lutz and Ines Milkulski, Requirements discovery during the testing of safety-critical software, Proceedings of the 25th International Conference on Software Engineering, 2003, pp578-583

[25] Accelerating the release of Kernel related bug fixes, http://sunsolve.sun.com/pub-cgi/show.pl?target=kernel_patch

[26] Eckhard Limpert and Werner Stahel, Life is log-normal!, Swiss Federal Institute of Technology Zurich, http://www.inf.ethz.ch/personal/gut/lognormal/brochure.htm

[27] Ekhard Limpert, Werner Stahel, Markus Abbt, Log-normal Distributions across the Sciences: Keys and Clues, Bioscience, Vol. 51 No. 5, May 2001, pp341-352

[28] Christian Gut, Basic Mathematical Properties of the Lognormal Distribution, http://www.inf.ethz.ch/personal/gut/lognormal/maths.ps

[29] Joe Marasco, The Project Pyramid, IBM developerWorks, http://www-106.ibm.com/developerworks/rational/library/4291.html

[30] Katy Dickinson, Software Process Framework at Sun, ACM StandardView Vol. 4, No. 3, September 1996, pp161-165

[31] Overview of Solaris Patch System Testing and Performance Regression Testing, http://sunsolve.sun.com/pub-cgi/show.pl?target=patches/sys-and-perf-test

[32] http://sunsolve.sun.com/

[33] Accelerating the release of Kernel related bug fixes, http://sunsolve.sun.com/pub-cgi/show.pl?target=kernel_patch

[34] Tilmann Bruckhaus et al, ibid.

[35] Talha Javed et al, ibid.

[36] Nancy Leveson, Intent Specifications: An Approach to Building Human-Centered Specifications, IEEE Transactions on Software Engineering, January 2000.

[37] Tilmann Bruckhaus et al, ibid

[38] Ram Chillarege, ODC for Process Measurement, Analysis and Control, Proceedings, Fourth International Conference on Software Quality, ASQC Software Division, Oct 3-5, 1994.

[39] Ram Chillarege and Shriram Biyani, Identifying Risk using ODC Based Growth Models, Proceedings Fifth International Symposium on Software Reliability Engineering, November 1994.

[40] David N. Card, Learning from our Mistakes with Defect Causal Analysis, IEEE Software, Jan-Feb 1998.

[41] Ram Chillarege, ODC for Process Measurement, Analysis and Control, ibid.