WHEN A PRODUCT FEATURE BECOMES AN INDUSTRY: 
AN EXAMINATION OF THE HARDWARE AND 
SOFTWARE FAULT TOLERANCE INDUSTRY 

By Dennis G. Pratt 

B.A., Northwestern University (1983) 

SUBMITTED IN PARTIAL FULFILLMENT 
OF THE REQUIREMENTS FOR THE 
DEGREE OF 

MASTER OF SCIENCE IN MANAGEMENT 

at the 

MASSACHUSETTS INSTITUTE OF TECHNOLOGY 

June, 1986 

© Dennis G. Pratt, 1986 

The author hereby grants to M.I.T. permission 
to reproduce and to distribute copies 
of this thesis document in whole or in part. 

Signature of Author: 

Alfred P. Sloan School of Management 
May 16, 1986 

Certified by: 

M. Anthony Wong 
Thesis Supervisor 

Accepted by: 

Jeffrey A. Barks 
Associate Dean, Master's and Bachelor's Program 

Archives 

MASS. INST. TECH. 
JUN 2 8 1986 
LIBRARIES
WHEN A PRODUCT FEATURE BECOMES AN INDUSTRY:

AN EXAMINATION OF THE HARDWARE AND
SOFTWARE FAULT TOLERANCE INDUSTRY

by

Dennis G. Pratt

Submitted to the Alfred P. Sloan School of Management
on May 16, 1986, in partial fulfillment of the requirements
for the Degree of Master of Science in Management

ABSTRACT

This thesis uses the lessons from the hardware fault tolerant industry
to explore how a software fault tolerant startup might enter and capture a
large segment of the future commercial software fault tolerant industry.

The author's underlying belief is that an industry based on a single fea-
ture is only a transitory industry. The traditional computer manufacturers
are slowly incorporating the high degree of hardware fault tolerance found
in the products of computer manufacturers that specialize in hardware fault
tolerance. As the traditional computer manufacturers become more successful in incorporating hardware fault tolerance, the need for a separate, highly fault tolerant computer industry will disappear. The implications for the software fault tolerant pioneer are important. If a pioneering software fault tolerant startup proves that software fault tolerance is also valuable to customers, the traditional computer manufacturers will again incorporate software fault tolerance into their products, obviating the need for the software fault tolerant company.

The slow absorption of a single important feature into the main industry implies that single-feature industries cannot sustain their competitive advantage. However, the single feature, if demanded by a large enough market segment and if not easily duplicated by the traditional manufacturers, does permit companies to enter an industry dominated by giants when the company otherwise could not do so.

Thesis Supervisor: Dr. Anthony Wong

Title: Associate Professor of Management
1 Acknowledgement

I would like to thank:

• my thesis advisor, Tony Wong, for his flexibility and entrepreneurship;

• my thesis reader, David Anderson, for his lack of demands;

• my Northwestern advisor, Krishna Kant, for his role in pointing out this subject;

• my Uptime fellow-entrepreneurs, Willie Lim and Rick Harper, for their insights, late nights, and hard work;

• my Sloan classmates, professors, and staff who have made my brief stay here so enjoyable;

• my wife, Carol Pratt, for herself; and,

• my late-night companion, Genie.
# Contents

1 Acknowledgement 4

2 Introduction 8

3 Single Feature: Short Term Advantage 10

3.1 HFT Industry Life Cycle 10

3.2 Transience of HFT Industry 12

3.3 SFT Industry Life Cycle 15

4 HFT: Classic Segmentation 19

4.1 Market Segmentation: A Proven Strategy 19

4.2 Segmentation in the Computer Industry 22

5 An Explanation of Fault Tolerance 27

5.1 Definitions of Fault Tolerant Terms 27

5.2 Triple Modular Redundancy for HFT 30

6 SFT: A Harder Problem 34

6.1 SFT: More Important than HFT 35

6.2 SFT in Future Systems 37
### 7 Approaches to Software Reliability

7.1 SFT Through Design Diversity .................................. 42
7.2 Fault Tolerant Notation ........................................... 45
7.3 Independence of Specifications ................................. 48
7.4 Concerns About N Version Programming ..................... 50

### 8 Examples of HFT

8.1 Inventory Control ................................................. 57
8.2 Banking .......................................................... 58
8.3 Point-of-Sale ..................................................... 60
8.4 Other Examples .................................................. 61

### 9 Examples of SFT

9.1 Government Software Faults ................................. 65
9.2 A Commercial Software Fault ............................... 68
9.3 Why SFT is Not Commercially Available .................. 69

### 10 Roots of HFT

10.1 Telephone Switching Systems .............................. 74
10.2 Real-Time Monitoring and Control Systems ................. 74
10.3 Airline Reservation Systems .......................... 76

11 Tandem: The Pioneer HFT Company .................. 78

11.1 Tandem’s Susceptibility ............................... 81

12 Enter the Startups ....................................... 87

12.1 Industrial Automation Startups ....................... 87

12.2 On-Line Trasaction Processing Startups ............... 91

12.3 Reasons for the Poor Performance ................... 98

13 Stratus: A Successful Startup ......................... 101

13.1 Stratus’s Architecture ............................... 104

13.2 CCI: Another Success? ............................... 106

13.3 Lessons from Startups ............................... 107

14 Role of VAR ............................................ 110

14.1 Stratus’s Lead with the Vars ....................... 112

15 Conclusion ............................................. 117

15.1 HFT Conclusions ..................................... 117

15.2 SFT Conclusions ..................................... 121
2 Introduction

Fault tolerance is a feature of a computing system, similar to other computer system features such as power, speed, user-friendliness, compatibility, programmability, or graphics capability. All computer systems have this feature to some degree. Computer hardware manufacturers that do not consider themselves "fault tolerant" boast of systems with less than 2% downtime. Many software companies have never had a software failure in their released software products.

However, since 1976 a commercial computer manufacturer has sold its systems mainly on excellence in one feature — hardware fault tolerance. The availability of a computer system that would not go down, even for a short period of time, due to a hardware failure, allowed customers to automate increasingly more important operations; the availability of the product created its own demand.

The large profits which this pioneer of hardware fault tolerant systems made encouraged twelve startups to attempt to develop products with similarly high hardware fault tolerance. The result has been the development of a new industry based solely on the hardware fault tolerant computer.
Software fault tolerance, on the other hand, has not yet been commercialized. It remains solely in the domain of government systems where failure would cause loss of life — usually only applied to defense, power plants, and aeronautical systems. The software fault tolerant industry is approximately at the same stage of development as the hardware fault tolerant industry of the 1960's. We can expect the development of software fault tolerance to mirror the development of the hardware fault tolerant industry.
3 Single Feature: Short Term Advantage

3.1 HFT Industry Life Cycle

The life cycle of the hardware fault tolerance industry has been as follows:

Pre-1960 Early theoretical interest in hardware fault tolerance.

1960-1970 Increasing hardware component reliability obviates the need for hardware fault tolerance for most applications. Interest in hardware fault tolerance relegated mainly to government and defense computer systems.

1970-1975 Automation of increasingly important commercial systems requires higher degree of fault tolerance. Traditional manufacturers try to upgrade their products to provide more hardware fault tolerance. Defense and aeronautical applications widely use hardware fault tolerance.

1975-1978 The pioneer hardware fault tolerant company, Tandem Computers, demonstrates the benefits of having a computing resource with virtually no downtime.
1978-1980 Tandem generates high ROI. End-user companies begin to automate even more important company functions, expanding the market for hardware fault tolerant computers.

1980-1983 New firms, attracted to the fault tolerant market by the high ROI, attempt to develop a better product for the market. Tandem continues to collect above average ROI.


1985-1988 Traditional computer manufacturers form joint ventures with hardware fault tolerant companies and begin intensive internal efforts to apply hardware fault tolerance to their own systems.

1988 On Traditional computer manufacturers recapture most of new hardware fault tolerant sales. Few opportunities left for specialized hardware fault tolerant manufacturers. Traditional manufacturers acquire hardware fault tolerant companies. Hardware fault tolerant manufacturers go out of business if not associated with a larger, traditional
3.2 Transience of HFT Industry

The foregoing implies that the computer hardware fault tolerant industry is actually a temporary sub-industry within the larger minicomputer industry. The difference between the offerings of the hardware fault tolerant companies and the traditional minicomputer companies will disappear as hardware components continue to become more reliable and as traditional manufacturers add many of the fault-tolerance-enhancing features to their basic model design.

The hardware fault tolerant computer manufacturers’ final product to the end-user is not significantly different from the traditional minicomputer manufacturers’ products; both the hardware fault tolerant manufacturer and the conventional minicomputer manufacturer offer approximately the same business solution. However, by stressing one key feature, fault tolerant manufacturers have carved out a relatively small ($700 million\(^1\)) niche in

\(^1\)1985 Revenues: Tandem $624M; Stratus $75M
a very large ($27 billion² total on-line transaction processing hardware) market.

Customers are the first to recognize that hardware fault tolerance is but a single feature of a computer system, a feature which is unable to sustain an entire industry.

"We purchase computer systems because of how well they solve our business problems, not because of any hardware feature. We don't care what features the hardware has, only whether the final system satisfies our needs. We don't care whether the system has hardware fault tolerance or not."³

Even the hardware fault tolerant computer manufacturers have recognized that fault tolerance is only one of many features, one which by itself cannot sell systems.

"Customers are no longer interested in hardware or system software. Today they demand total system solutions to business

²[2]
³[1]
The fault tolerant companies might sell the customer on the need for fault tolerance, however, fault tolerance is rarely the primary concern.

"Once we decided to have fault tolerance, fault tolerance was a requirement, but we made our decision (to purchase a Stratus hardware fault tolerant computer) not on the degree of fault tolerance, but on how well the system could deliver us our service."\(^5\)

Hardware fault tolerant companies recognize that their short term advantage is slowly ebbing. With IBM's entrance into the fault tolerant market through its joint venture with Stratus, that advantage has all but disappeared.

The hardware fault tolerant industry will someday be an interesting historical footnote; an industry which was conceived in an era when computers failed frequently and which died quickly once hardware became more

\(^4\) [3], p.4

\(^5\) [4]
reliable. The hardware fault tolerant manufacturers seek to make large inroads into the highly critical, automated markets which their products have made possible. Their hope is that, by the time they enter, the fault tolerant companies will have developed business solutions for these markets which are better, and more proven, than those that the traditional manufacturers can provide.

3.3 SFT Industry Life Cycle

While the software fault tolerant industry has yet to become a separately recognized subindustry, we can anticipate that its life cycle will mirror the experience of the hardware fault tolerant industry. To date its life cycle has been as follows:

1960–1970 Little commercial interest in software faults — companies view software faults as an inherent part of the computer system. Most emphasis on software design faults is on testing and assurance to find and eliminate most of the faults. Interest in software fault tolerance amongst the defense industry.


The improvement in software engineering methods is analogous to the increased reliability of hardware components during the 1960's. The increased hardware reliability was a stop-gap measure for systems which eventually became so complex and so critical as to require hardware fault tolerance. The current software lead-users (e.g. defense) deep concern with software fault tolerance is similarly analogous to the hardware lead-users (also defense) in the early 1970's demand for high degrees of hardware fault
tolerance. A recent Fortune article\(^6\) reported:

The (software) situation is critical in the military where software problems are not just a matter of lost opportunity, but a potential danger to national security. 80% of the US weapons systems in development depend significantly on software.

The Pentagon and its contractors are already having trouble fielding weapons with relatively simple software.

Software is about the biggest stumbling block to President Reagan’s “Star Wars” anti-missile system. David Parnas, a prominent computer scientist, resigned after concluding that reliable software for Star Wars could never be built. Most other experts working on SDI believe reliable software can be developed with enough time and research money, but only if planners make software the paramount technical priority.

If software fault tolerance mirrors hardware fault tolerance, we should expect a software fault tolerant pioneer to commercialize software fault

\(^6\) [5] p.132
tolerance once commercial software systems become as complex and critical as current defense software.
4 HFT: Classic Segmentation

Although what the hardware fault tolerance companies have done has allowed them only a temporary advantage, their strategy has been rather clever. It is the essence of business. They have located some very advanced customers for whom the existing 98% availability of traditional computer systems is not good enough. These customers’ applications require almost 100% availability of the computer resources; severe economic penalties accrue to these leading-edge companies if they lose their computer resources.

4.1 Market Segmentation: A Proven Strategy

Finding a high-need segment of customers whose need is not currently filled by existing products is the key first step of successful new product marketing. While marketers use sophisticated techniques to identify new market segments in old industries, most of these techniques are applied only to consumer goods markets. For example, marketing identified new market segments within the toothpaste industry including the pump for those people who hate to deal with messy tubes, and special toothpaste for smokers, for anti-tartar, for regular brightening, and for various taste sensations.
While advanced marketing techniques are sometimes used as part of a post hoc analysis of the existing computer market, these advanced techniques have not been applied widely to new computer product design. Why have advanced marketing techniques not infiltrated the computer industry? Some possible reasons are:

- While every marketer can understand toothpaste, not all marketers have the required technical backgrounds to understand the nuances of hardware and software engineering.

- The culture of existing computer companies supports the view that new products are developed in a lab by "external forces be damned" engineers working diligently and covertly on their pet project. For example, in Soul of the New Machine\textsuperscript{7}, Tracy Kidder describes how Tom West and the "microkids" secretly put together a 32-bit superminicomputer more powerful than anything on the market despite many people, both inside and outside the company, actively trying to undermine their efforts.

\textsuperscript{7}[6]
In most computer companies “marketing” is synonomous with “sales.” Few computer companies have marketing departments actively involved in new product design. Because they seek salesmen for “marketing” positions, computer companies regularly have salesmen in marketing positions. The salesmen have little interest in applying or ability to apply new product marketing to their company’s products. This produces a vicious cycle; computer companies do not understand the value of hiring new product marketers because their current “marketers” (i.e. salesmen) cannot help design new products,

A common misconception is that there is only one important dimension for computers — power. Engineers regularly complain to marketers that they know well what the customers want — more power per dollar. This statement is akin to saying that customers want only cavity protection from their toothpaste.
4.2 Segmentation in the Computer Industry

A good example of traditional (i.e. single dimension) new computer product innovation is Cray Computers\textsuperscript{8}. Seymour Cray was an eccentric IBM engineer who wanted to build superfast computers. When IBM balked at implementing Mr. Cray’s dream in favor of serving the broader market, Mr. Cray started his company. Cray Computers has discovered a small segment of users who are willing to pay disproportionately higher prices for systems which can run marginally faster than the fastest IBM computers.

Weather forecasting agencies are some of Cray’s customers. These agencies must run extremely sophisticated weather models with tens of thousands of inputs coming in every hour. The more data per hour that the weather agency can enter into the weather models, the better the forecast. However, they must have finished the previous hour’s forecast before the next hour’s input comes in. Eventually, the weather forecasters would like to have millions of inputs into an even more sophisticated model, with the inputs arriving more frequently. The needs of a customer like this are entirely different from the needs of other high end computer users. This

\textsuperscript{8}[7]
application cannot use the plain vanilla high-end computer. It calls for as
much speed as possible, and Cray provides that speed.

Both Cray and the hardware fault tolerance companies identified a niche
whose requirements for a single feature were much more demanding than
the requirements of the broader computer market for that feature. Cray fo-
cused on the traditional speed/power dimension, whereas the fault tolerant
companies identified a different attribute in which to excel.

A concern which Cray and the fault tolerant computer manufacturers
share is whether the niche they have identified is permanent. The single
processor supercomputers are quickly approaching their theoretical speed
limits. At the same time, traditional computer manufacturers are making
their computers much faster, narrowing the difference between their offer-
ings and Cray Computer's offering. Parallel processing computers promise
to leapfrog the speed gains of the Cray single processor approach.

While Cray Computers's products exemplify the typical engineer's idea
of a product innovation, other examples belie the single dimension (i.e.
power) approach to segmentation in the computer industry; other dimen-
sions have been the source of profitable niches in the once monolithic com-

24
puter market. These niches seem to have been discovered almost by accident. Usually, the computer company's culture tells the tale of the engineer who "just knew" he was onto something.

At the low power end, Jobs and Wozniak noticed that hackers (like themselves) were tired of shared mainframes and were experimenting with small dedicated computers. They packaged preexisting hardware, gave it a cute name, sold intensively, and an American legend was born.

When IBM entered the microcomputer market, IBM unremarkably decided that power would be the predominate dimension for microcomputers. Their PC contained more memory and had a faster processor than the Apple IIe. Apple's response was not to fight IBM head on, but to hit IBM where IBM had been susceptible all along. The Macintosh is the easiest to use microcomputer available. IBM was correct to assume that power would sell the majority of the market. But while business tends to buy power, a large number of users prefer the graphics and simplicity of the Macintosh. Apple is still profitable four years after IBM's entry.

Minicomputers designed for their graphics capability have captured another segment of the computer market. Companies such as Apollo and Sun
Microsystems stand out as successfully capturing a large part of a small niche. These companies' dedicated workstations enabled great strides in computer-aided design (CAD), manufacturing (CAM), engineering (CAE), and, recently, software engineering (CASE).

Symbolic processor manufacturers such as Symbolics and Lisp Machines Inc. (LMI) have also identified and captured a niche market. Most computers were designed according to the Von Neuman architecture. This historical architecture has severely restricted the growth of programming languages. The (procedural) languages currently available, such as Fortran and Pascal, are not optimal languages for many applications. The severe hardware architectural constraints of the Von Neuman architecture prevented more appropriate languages.

Newer languages, especially the functional languages of Lisp and Prolog, cannot run optimally on the current hardware architecture. Symbolics and LMI have entered the computer industry to provide users with computers which run Lisp better than any traditional computer.

Future niches include parallel processors in which much of the processing
occurs simultaneously instead of sequentially, and neural nets which are able to solve a large class of NP complete problems\textsuperscript{10} in less than exponential time.

Hardware fault tolerant manufacturers have similarly discovered a small market niche well off the traditional power path. By stressing a single feature — marginally reduced downtime — they have created an entire industry.

\textsuperscript{10}NP complete implies a problem whose running time is exponential to the number of inputs (e.g. the traveling salesman problem).
5 An Explanation of Fault Tolerance

An important aspect of a computer system is the degree to which the user can rely on the system to perform its intended function. Early theoretical work in fault tolerance emphasized handling physical (i.e. hardware) faults. Recent theoretical emphasis has been placed on tolerating design faults occurring in both hardware and software.

5.1 Definitions of Fault Tolerant Terms

The concepts of "failures", "errors", and "faults" are associated but not identical. A failure occurs when the user loses the service of a computer system resource (e.g. the user can no longer access the central computer from her terminal). It is not important that we know what caused the system failure — explanations could range from a crash of the central computer's CPU to an electrician cutting the cables connecting the terminal to the central computer. The key issue is that the user no longer has the

\[^{11}\text{\cite{9} pp. 68–69}\]
service of a computer resource\textsuperscript{12}, therefore, the system has experienced a failure.

An \textit{error} occurs when some part of the computer system resource assumes an undesired state. If a memory location contains an incorrect value, an error has occurred in the memory resource. The incorrect value could lead to a failure. For example, the incorrect value could cause the destruction of the disk controller software, thus preventing the user from accessing the disk. However, an error in itself is not a failure. Here again, we do not know what caused the error. Plausible hypotheses could include a bug in the software or a physical fault in the CPU.

The (hypothesized) cause of the failure or error is said to be a \textit{fault}. We could hypothesize that the reason for the user’s inability to access the central computer (i.e. for the failure) was because of a fault in the CPU. Likewise, we could hypothesize that the reason for the incorrect value in the memory location (i.e. for the error) was a software fault which incorrectly placed the value at that location.

\textsuperscript{12}In this case, the “resource” is the central computer, but we could construct examples in which the resource is any part of the computer system, e.g. a disk drive or a communication channel.
A *hardware physical fault* is an incident in which a hardware component of the computing system fails due to random physical phenomena. For example, if the power supply for the computing system fails, the failure of the computer system would be said to be caused by a hardware physical fault.

*Design faults* are faults which occur not because of the age of the system or because of environmental or physical factors. Instead, the design fault usually is due to a (human) logical error which causes the system to act differently than the user expects. Design faults can occur in both hardware and software subsystems. Design faults become incorporated unintentionally into the computing system at the time the software program (or the hardware) is being designed and are not discovered by the designers during the testing and acceptance phase. The software program (or hardware component) is released to the customer. The design fault lays in wait for the set of circumstances which enables it to cause either an error or a failure.

Because design faults are more likely to occur in software than in hardware, most theoretical discussion concerning design fault correction has
centered on software. As logic is increasingly being packed into hardware components, the probability of incorporating design faults into hardware has increased. Nevertheless, hardware design fault tolerance is still not a major thrust of fault tolerance. Thus, hardware faults usually refers to hardware physical, not hardware design, faults.

Fault tolerance is the ability of the computer system to survive a fault. The goal of fault tolerance is for the system to recover automatically from the fault-caused error, and to eliminate the fault without the user experiencing a failure. Mechanisms for implementing fault tolerance include redundant hardware, redundant software, and elaborate algorithms for detecting faults and recovering from them.

5.2 Triple Modular Redundancy for HFT

The key to hardware physical fault tolerance is to eliminate single points of failure. For example, if the computing system contains only one CPU, that CPU would be a single point of failure since if the CPU failed, the

\(^{13}\)For the sake of brevity, "software design faults" is shortened to software faults, and "hardware physical fault" to hardware fault.
entire computing system would fail. Other likely single points of failure in traditional computing systems include buses, internal memory, and external memory.

Currently the "best" theoretical solution to hardware fault tolerance is called *N modular redundancy*, with *N*>=2. In this approach, *N* separate, but identical, hardware subsystems execute identical copies of the software program. The output of each hardware subsystem is checked periodically against the output of the other *N*-1 subsystems. This theory assumes that if one of the hardware subsystems experiences a (random) hardware fault, it is improbable that the other hardware subsystems will simultaneously experience the same hardware fault. The assumption that hardware faults will occur independently is central to providing hardware fault tolerance.

If *N*=2 and a fault occurs in one of the hardware subsystems, by checking the output of the subsystems, the computing system will be able to recognize that a fault occurred. (See Diagram 1). When *N*=2 the computer system cannot continue executing since the computing system cannot know which hardware subsystem is faulty and which is correct. At least the error is identified before some unknown, and possibly catastrophic, error or
failure occurs. For example, perhaps the faulty hardware subsystem would have closed a nuclear reactor water valve, accidentally causing a melt down. With an N modular redundancy scheme, with N=2, the valve is not closed and the error is reported to a human operator. Other corrective (usually manual) action can take place.

If N>2 then the computing system can continue operating despite a hardware failure or error in some of its subsystems. By comparing the output of each N hardware subsystem, the computing system can recognize when one of the subsystems is faulting. The computing system basically uses a majority vote algorithm to decide which subsystem is incorrect. By assuming that the majority of the subsystems are producing the correct output, the computing system can shut down the M faulting hardware subsystem(s) and continue executing its task in an N-M modular redundancy mode.

Because of the ability of an N modular redundancy systems to mask faults when N>2, and because cost constraints dictate adding the least amount of hardware possible, N is frequently chosen to be three. (See Diagram 2.) This widespread system is called *Triple Modular Redundancy*.
(TMR). In our nuclear reactor example, the output from CPU A and CPU B agree with each other, but disagree with CPU C. The computing system assumes that CPU A and CPU B are correct, disconnects CPU C, and continues processing using only CPU A and CPU B.
6 SFT: A Harder Problem

"As the hardware becomes more reliable, software faults — already recognized as being more of a problem than hardware faults in many situations — will become even more visible.

"An intriguing idea is that software creation might be brought up to a fault-free level comparable to that of hardware design if software designers had available the equivalent of computer chips — small, well-tested, widely used off-the-shelf packages.

"The design of reliable software is very much an intractable problem today. Whoever solves it will gain a well-deserved place of honor in the history of computing."14

Software methodology has not developed to the point where software firms can assure customers that their code is bug-free (although a surprising number of software firms do pretend to give that assurance.) Software that produces incorrect results can have consequences equally as serious as a hardware fault.

14[10], pp. 29–30
6.1 SFT: More Important than HFT

An AT&T\textsuperscript{15} analysis of the faults causing errors or failures on their 3B20D computer indicated the following incidence of faults:

35\% Application Software Faults

30\% Operator Faults

20\% Hardware Faults

15\% Operating System Software Faults

Thus a full 80\% of faults were not hardware faults. 50\% of the faults were due to both operating system and application software design faults and almost a third of the errors were due to operator faults. Neither of the latter types of faults are addressed by the current fault tolerant companies.

In a similar study Security Pacific Automation Corp\textsuperscript{16} reported\textsuperscript{17} the following sources of failures for their three IBM mainframes:

\textsuperscript{15}[10], p.28

\textsuperscript{16}The data processing operation of California’s Security Pacific Bank

\textsuperscript{17}[11], p.7
49.7% Application Software Faults

24.0% User Errors

6.7% Incorrect User Invoked JCL

5.6% Incorrect Software Invoked JCL

5.1% Tape Drives

5.1% Bad Tapes

4.6% Space Problems

3.4% Operating System Software

2.8% Software Security

2.3% Operator Error

0.6% CPU Failure

Because Security Pacific actually disaggregated the types of hardware faults, the Security Pacific data are more interesting than the AT&T study. CPU failure, which is the predominant focus of the commercial hardware
fault tolerant computer manufacturers, caused less than one percent of system failures. Even when we include tape drives and tapes in the hardware figures — items for which the fault tolerant manufacturers do not explicitly correct — the amount of system uptime which would be increased by using current hardware fault tolerant products would only be 11%.

On the other hand, software faults, including application, operating system, security, and application JCL software, account for 61.5% of system failures. A quarter of the failures, 25.3% occurred from operator and user errors — something extremely difficult to protect against and totally ignored by current manufacturers.

Thus we see that software faults and operator faults cause more downtime than hardware faults and that none of the current hardware fault tolerant companies address these more important sources of faults.

6.2 SFT in Future Systems

"One of the drawbacks of a computer with many processors is the increased opportunity for both hardware and software malfunctions. Today's single-processor machines can run
without a major hardware fault for up to a few thousand hours.

Software errors can take place within tens or hundreds of hours.

At such rates, a 1,000-processor system would have a hardware failure every few hours and a software failure every few minutes.

We cannot allow a single failure to cause an entire multiprocessor system to crash.

"...Some researchers are postponing work on (software fault tolerance) until they solve what they feel are more pressing architectural and programming issues (of multiprocessor systems).

I believe this is a misguided approach, since (software) reliability may turn out to be pivotal to the evolution of multiprocessors.  

As the Dertouzos quote illustrates, many of the newer applications such as multiple processor systems will be hamstrung until software fault tolerance is solved. Dertouzos gives an idea of the relative importance of hardware versus software faults. He believes that software faults occur one to two orders of magnitude more frequently than hardware faults. Given the relative frequency of faults, perhaps all the emphasis on hardware faults

---

\[16\] [12], p.45
has been misplaced.

One of the most serious arguments against the Strategic Defense Initiative (SDI) is that it is unclear how the large amount of enormously complex software that SDI requires can be made sufficiently fault-free to assure its success\textsuperscript{19}.

The software specification document for a project as complex as SDI will itself require tens of thousands of pages.

"The possibility that all potential situations would be adequately accounted for in this massive document is at least as likely as the possibility that no unanticipated loopholes exist in the (3000) page Federal Tax Laws."\textsuperscript{20}.

In addition, by translating the specifications to code, design errors will be added to the already flawed specifications\textsuperscript{21}. The system will require multiple, concurrent processes\textsuperscript{22} with deadlines which, if violated, would

\textsuperscript{19}[13], p.1
\textsuperscript{20}[14], p.48
\textsuperscript{21}[15]
\textsuperscript{22}A process describes the user program when it is running in the computer system.
cause irreparable damage.

Testing the programs will be of limited use since many of the software faults would not be observed except under the heavy load and "unknown unknowns" of actual usage. A full blown simulation would still not assure the reliability since many software errors are "soft"\textsuperscript{23} and since a simulation must follow rules which a real attack might ignore.

Again we see that at the leading edge of computer applications, software reliability is an extremely important issue, and one that, while addressed somewhat in theoretical works, is not applied at all to commercial applications.

\textsuperscript{23}I.E. Occuring only under certain timing situations.
7 Approaches to Software Reliability

Anderson\textsuperscript{24} calls the two main thrusts to construct reliable software "fault intolerance" and "fault tolerance". \textit{Fault intolerance} (also known as fault elimination or fault prevention) describes a class of techniques in which an attempt is made to ensure that the software contains no faults. These techniques include: requirements definitions, precise specifications, design and programming methodologies, program proving, and testing and assurance. These techniques, while reducing the number of faults, cannot completely eliminate faults.

\textit{Software fault tolerance} describes a class of techniques in which the focus is on providing acceptable service despite the (inevitable) existence of software design faults. The main methods used for software design fault tolerance are recovery blocks and N version programming.

With current commercial hardware fault tolerant systems, hardware fault tolerance is achieved through replicating exact copies of the hardware components. Output from these components are compared against each

\textsuperscript{24}\cite{20}, p. 353
other. Should the output from the exact components not agree, a physical fault is assumed to have occurred.

This approach has only limited applicability to software. Some researchers\textsuperscript{25} have suggested that a recovery block mechanism that reloads a new copy of the faulting software might increase reliability. This approach only catches transient errors which may have corrupted the original copy of the software. This transient software fault correction strategy is used with some Stratus systems.

7.1 SFT Through Design Diversity

While it is important to correct transient faults, design faults caused by human oversights and logical mistakes are also potentially dangerous. These design faults occur during specification, design, operation, updating, and maintenance of the hardware and software of the computing system\textsuperscript{26}. To cope with design faults, the computer system must contain multiple, independent versions of the program.

\textsuperscript{25}[17]

\textsuperscript{26}[16]
Design diversity\textsuperscript{27} is the approach in which the software elements that are to provide fault tolerance are not exact replicas, but are independently designed to meet a system's requirements. Different designers and design tools are employed in each effort, and commonalitites are systematically avoided. The obvious advantage of design diversity is that computing may be made reliable without requiring the complete absence of design faults; the only requirement is that those faults not produce similar errors in a majority of the designs\textsuperscript{28}.

"Design faults will produce identical errors in all computations if identical copies of hardware and software elements are employed to execute multiple computations that have the same inputs and initial values. Such total susceptibility to design faults is the most serious limitation of the multiple computation approach."\textsuperscript{29}

\textsuperscript{27}[9], p.67
\textsuperscript{28}[9], p.68
\textsuperscript{29}[9], p.69
The two major approaches to SFT using design diversity are Recovery Blocks and Parallel Versions.

In the recovery block scheme\textsuperscript{30} a single hardware subsystem\textsuperscript{31} executes a job. At strategic points while the program is running, the most recent (intermediate) results of the software program are checked by an acceptance test to determine whether the results are correct so far. If the acceptance test fails, the system assumes that a software fault has occurred. The software program is rolled back to a prior state in which the software program had provided correct results. A new, independently designed, version of the software is loaded into the computing system. The computing system then tries again to complete the job.

Three key problems exist with the recovery block approach. First, the output of the software could pass the acceptance test but still be incorrect. In this approach, your fault tolerance is only as good as your acceptance test. In fact, the acceptance test itself could have bugs in it. Second, many real-time computer systems must complete their task by a certain deadline.

\textsuperscript{30}[17]

\textsuperscript{31}e.g. a single processor computer.
The roll back to a prior state and the reprocessing using the new program takes quite a bit of time; the deadline could easily be passed. Third, the roll back itself is difficult to implement.

With the rise of parallel processing, many authors\textsuperscript{32} have suggested running the N separate software versions in parallel on N separate hardware subsystems. At key points, the output of the different, independently designed software can be compared. The approach is very similar to the TMR approach for hardware except that in TMR each of the hardware subsystems is identical to the others; here, each of the N versions were written by different teams of programmers. Should the output from the N versions disagree, a majority vote is taken. The computing system disconnects the incorrect versions and continues processing.

7.2 Fault Tolerant Notation

Multiple computations can occur in three domains: time (the number of times the same copy is replicated); hardware subsystems (the number of

\textsuperscript{32}For example, see [21].
processors used); and program (the number of copies of software)\textsuperscript{33}. A handy notation for specifying different types of systems is: “\(xT/yH/zS\)” where “T” refers to time, “H” to hardware systems, “S” to software programs, and “x”, “y”, and “z” are integers. Most systems use a 1T/1H/1S (or “simplex”) system where one copy of a program runs on one hardware device one time.

Both the “H” and the “S” may take a “D” as a modifier, with the “D” standing for “diverse” (i.e. independently designed). Thus, if we had a rollback mechanism which used two independently designed implementations of the software running on one hardware device, we would have a system denoted by 1T/1H/2DS. Here are some frequently used systems:

1T/1H/1S Most computer systems use this system. Provides no fault tolerance.

2T/1H/1S Handles transient errors by reloading the software and trying again. Seen on systems with a recovery block scheme but with only one software program.

\textsuperscript{33}[9]
**1T/4H/4S** NASA's Space Shuttle runs four IBM computers. Each computer has own identical copy of the software.

**1T/3H/3S** Draper Laboratory's fault tolerant multiprocessor. Three identical processors run identical software copies.

**1T/2H/2S** Bell Labs 3B20D computer. Similar to Draper's configuration, but cannot mask a "soft" fault (i.e. a fault which, while it does not bring the system down, it causes the system to output incorrect data.)

**2T/2H/2S** Tandem's NonStop Line. One processor runs a copy of the software, if a hardware fault occurs, a second processor runs an identical copy of the software.

**2T/1H/2DS** A recovery block scheme, this time using different software after the system is rolled back.

**1T/3H/3DS** N version programming with N=3. All diversely designed copies of the software run in parallel.

**1T/2DH/2S** Two identical software programs run on two separately designed hardware processors.
**1T/3DH/3DS** Three (diverse) version copies of the program run in parallel on three (diverse) version hardware.

The diversely designed software programs improves the software fault tolerance. The key assumption of N version programming is that the design faults of one version of the program will not be duplicated in the other versions. To assure this independence of design faults, each version must be written in isolation from one another; the project leader cannot allow any communication between the different version teams.

### 7.3 Independence of Specifications

While multiple, independent programs may be written to execute the function, a basic problem with the independence assumption is that usually all programs are written from the same specification. The specification itself may contains faults.

Avizienis\(^{34}\) implemented an experiment in multi-version programming in which they used three different specification techniques. The specifi-

\(^{34}[16]\)
cations were written in the formal specification language OBJ\textsuperscript{35}, in the nonformal PDL (Program Definition Language)\textsuperscript{36}, and in English\textsuperscript{37}. Five of the resulting faults occurring in the versions were traced to the specifications: one specification fault occurred in the OBJ version, four in the English version, and none in the PDL version. The multiple specifications approach used here assures some additional software fault tolerance. But since the same team developed all three specification versions, the specifications were different only in the languages used; we could not say that they were developed independently.

Ramamoorthy et al.\textsuperscript{38} went a step further when they wrote two independent specifications in a formal specification language. They used formal mathematical techniques to verify that the two specifications were consistent with one another before continuing.

A final step might be to attempt to write multiple independent specifications in different formal specification languages. We could thus have

\textsuperscript{35}Developed at UCLA

\textsuperscript{36}PDL is an industry standard

\textsuperscript{37}An American standard.

\textsuperscript{38}[18]
independence in the specification design and in the specification syntax.

The formal specifications could then be tested against each other. Using a formal specification language is important since:

"A formal specification has precisely defined syntax and semantics. They can be studied mathematically, can be mechanized and tested to gather empirical evidence of their correctness, and can be computer processed to remove ambiguities, to eliminate inconsistencies, and to test empirically."\(^{39}\)

7.4 Concerns About N Version Programming

"It is assumed in some analyses of the (N Version Programming) technique that the N different versions will fail independently; that is, faults in the different versions occur at random and are unrelated."\(^{40}\)

If the independence assumption holds, the probability of failure of the

\(^{39}\)[9],p.72

\(^{40}\)[19],p.96
system is\(^{41}\):

\[
\Pr(\text{Failure of System}) = K \times \Pr(Vi) \times N
\]

Where \(K\) is a constant, \(N\) is the number of versions, and \(Vi\) is the probability of one of the versions failing independently. Thus, the probability of failure of an \(N\) version system, if the versions contain independent faults, is much lower than the probability of failure of any of the individual versions. However, even small probabilities of coincident errors cause a substantial reduction in reliability.

In addition to the added theoretical reliability of \(N\) Version Programming many researchers\(^{42}\) have suggested that the cost of the additional design and development time to produce \(N\) version software may be defrayed if the versions are tested against each other instead of using the normal, elaborate testing and assurance procedures for each version. The independence assumption is again crucial because this approach would only work if the versions contained independent faults.

\(^{41}[19], p.97\)

\(^{42}[9], p.79\)
Knight\textsuperscript{43} has provided extremely important data that the assumption of independence, so critical for N version programming, is not valid. Knight worried that “when solving a difficult intellectual problem, people tend to make the same mistakes even when they are working independently.”\textsuperscript{44} They felt that a small subset of the problems in any program were much harder than the rest of the problems, and were more likely to cause programmers to err.

They asked 27 programmers from two different universities to program independently the “launch interceptor” problem. This problem presents some rather sophisticated geometric problems. Each programmer tested her program with fifteen test cases. After the program passed these tests, the programmers could devise any other tests they wanted to run on their programs. The programs were collected at a central site where 200 additional random tests were run on each version. If a program passed all these tests, it was accepted as part of the study.

Each of the 27 versions were then tested on one million randomly gener-

\textsuperscript{43}[19]

\textsuperscript{44}[19], p.96
ated test cases. Knight compared the results of a “gold program”, known to
be highly reliable, to the results of the 27 versions. While the versions had
different failure rates (six programs had zero failures while six programs
had over 800 failures (with 9656 failures the maximum)), up to eight pro-
grams failed simultaneously on the same input. Knight and Leveson were
able to reject the null hypothesis (that each version had an independent
probability of failure) with a 99 percent confidence level. They thus were
able to reject the assumption of independent failures.

In a more detailed analysis of the faults, they were able to detect 45
faults in the program versions. Some of the same faults occurred in more
than one version. Examples of faults which did not occur in more than one
version were the omission of the assignment of a value to a function for one
path through the function, and the use of the wrong expression to index
an array. Examples of faults which did occur in more than one version
were assuming that limited precision comparison of the cosines of angles is
equivalent to the comparison of those angles, and neglecting to test both
zero and 180 degrees to identify a straight line. Thus, the faults that N
version programming might be able to identify seem to be straightforward
programming logic errors and typos. However, the faults which are not independent seem to involve more basic misunderstandings of the harder aspects of the domain of the problem (in this case, geometry.)

Knight suggests that

"the reliability of an N version system may not be as high as theory predicts under the assumption of independence. . . . Certain parts of any problem are just more difficult than others and will lead to the same faults by different programmers. . . . the faults. . . . involve boundary conditions or unusual cases. . . . This suggests that the fault distribution is more an artifact of the problem itself than (of) the programmer.

"Another possible hypothesis is that unique faults tend to be those more likely to be caught by a compiler or by testing. Common faults may reflect inherently difficult semantic aspects of the problem or typical human misconceptions which are not easily detected through standard verification and validation efforts."\(^{45}\)

\(^{45}\)\[19\], p.103
Knight and Leveson note that N Version programming may be used to correct many types of software faults, but that its fantastic theoretical reduction of faults may not be attainable in real systems.
8 Examples of HFT

Many key applications require the eradication of both physical faults and design faults. A nuclear power plant illustrates the need for both hardware and software fault tolerance. The potential of a software or a hardware fault causing a failure resulting in the death of millions of people has prompted nuclear power plants to use both hardware and software fault tolerant schemes.

Horror stories abound of dire consequences due to hardware failures. Perhaps the most potentially serious one was in June, 1980 when a computer circuit for NORAD failed\(^{46}\). The NORAD system to report that the United States was under nuclear missile attack from the Soviet Union, placing the entire US armed forces on alert, preparing the US nuclear missiles and scrambling B-1 bombers into the air to begin the counter attack. Fortunately, the defense department believes in remote redundant sites. When the attack could not be verified via a second system, the retaliation was postponed.

\(^{46}\)[14], p.48
With the widespread commercial use of hardware fault tolerance, we can concentrate on the benefits of hardware fault tolerance, instead of the costs of not having hardware fault tolerance. One of the conclusions from talking to many users of commercial hardware fault tolerant computer systems is that companies are implementing more critical systems than were possible before hardware fault tolerance. Gillette in Andover is an example\textsuperscript{47}.

8.1 Inventory Control

When Gillette sought to replace its Nixdorf-based inventory control computer system, the management had no intention of implementing the extremely complex, automated system that they finally decided to implement. Hardware fault tolerant manufacturers convinced Gillette to automate their entire ordering and distribution system.

In the proposed system, due in 1987, salesmen enter customer orders from remote sites. The computer system then takes over. The computer checks the order against existing inventory, schedules delivery, sends loading orders to a terminal attached to the forklifts, adjusts inventory, and prints

\textsuperscript{47}[4]
the required reports.

In the past Gillette had always worried about the computer system failing and considered this degree of automation impossible. A failure of the computer system for this application would be extremely costly. It would cause the entire distribution system to halt entirely, delaying the product to customers, idling warehouse workers, and preventing new sales.

8.2 Banking

Bank of Boston\textsuperscript{48} employs a three category description for the criticality of its computing resources as follows:

\textbf{Critical} The resource must be able to continue operating despite catastrophic accidents, including total loss of the computing site. Examples include the main data centers and the international network system nodes.

\textbf{Important} Downtime of an hour to a day is permissible. Alternates to computer based transactions are available. An example is the ATM\textsuperscript{49}

\footnotetext[48]{[1]}

\footnotetext[49]{Automatic Teller Machines}
network.

**Less Critical** Downtime, while inconvenient, is not a main concern.

The most critical Bank of Boston computer system, and the one requiring the largest expense for hardware fault tolerance, is the data center where IBM 3090 class mainframes are employed. To achieve hardware fault tolerance, each data center periodically downloads its data to another "hot" backup data center. Bank of Boston is protected even should one of the data centers blow up; another data center would take over for the destroyed data center and Bank of Boston would lose no more than two hours of transactions.

Banking network systems also require a high degree of hardware fault tolerance. Bank of Boston uses hardware fault tolerance for the minicomputer nodes in Bank of Boston's international network. Central, extremely critical, nodes are located in Paris, Tokyo, and New York. In addition there are local nodes. In the New England area local nodes are located in Boston and Springfield. As William Synnott explained,

---

50 The hot backup data center executes its own jobs while being able to take over for the first data center should the first data center fail.
"We must have an action plan which encompasses a worst case scenario involving the total destruction of a node, whether due to employee malfeasance, war, or fire. I have to quickly reroute messages in the network, and I can't lose data."

The ATM network, is a much less critical computing system. If an ATM fails, customers have an alternate method for executing transactions — they can walk into the bank and deal directly with a teller. However, Bank of Boston would prefer that this system not go down frequently since a failed ATM inconveniences customers.

8.3 Point-of-Sale

Hardware fault tolerance permits a high degree of automation. Gillette is just one example. Mobil Oil's point of sale (POS) is another^51.

Mobil's Tandem-based POS system automatically authorizes credit and debit card purchases at Mobil's gas stations. Customers' accounts at Mobile and at local banks are automatically updated. In addition, customers can use their ATM card.

^51[3], p.3

61
The system benefits Mobil by replacing the manual handling of charge receipts which previously delayed accounting and billing by ten days. This improved system decreased Mobil's working capital requirements, allowing more throughput at their stations, and attracted customers by the added convenience of the system. Tandem estimates that in 1986 3,600 of Mobil's 12,000 service stations will have the POS system and that the system will handle seven million transactions per month.

8.4 Other Examples

Many state lottery systems use hardware fault tolerance. The lottery is an important source of state revenue. If the system should go down, the states could lose as much as $10K/hour of down-time depending on the time of day, the size of the lottery award, and the proximity to the lottery's deadline\textsuperscript{52}.

Many of the financial trading exchanges use fault tolerant computers to ensure continued trading despite hardware faults. The Chicago Board of Options' use of Stratus in its trade reporting system is an example.

\textsuperscript{52}[35]
NASA uses five IBM computers for hardware fault tolerance on the space shuttle. The architecture is N modular redundancy with N = 4. The fifth computer is in stand-by, waiting to be added should the shuttle crew desire it. (It would be added only if the four computers reached a two-two tie, or one of the four computers goes down.)
9 Examples of SFT

During the first space shuttle launching in April 1981 the NASA hardware fault tolerant system described above provided one of the most famous examples of the successful application of software fault tolerance\textsuperscript{53}. Strangely, software fault tolerance was not an intended feature of the NASA system. The software fault tolerance illustration occurred accidently, and perhaps saved lives.

During the launch sequence the system operator attempted to add the fifth computer to the four running computers. The fifth computer had an important difference from the other four computers: its operating system was written, not by IBM, but by Rockwell International. At one of the synchronization points twenty minutes before launch, the fifth computer reached a state which was inconsistent with the other four. The mission was postponed for two days.

The first four computers' operating system had had a software fault — a timing error which had a one in sixty-seven chance of occurring. Garman

\textsuperscript{53}[22]
describes that the bug was due to “a very small, very improbable, very intricate, and very old mistake\textsuperscript{54}.”

“It was the kind of mistake that can never be ruled out in the world of real systems development — a world involving hundreds of programmers and analysts, thousands of hours of testing and simulation, and millions of pages of design, specifications, implementation schedules and test plans, and reports\textsuperscript{55}.”

The error was actually introduced two years before the launch, during an attempt to fix another problem\textsuperscript{56}. Since the four computers were all running the same operating system, had the fifth computer not been on-line, the fault would never have been noticed — perhaps with tragic consequences. However, the separately written Rockwell operating system did not have the same software design fault. The fault was identified before failure occurred.

This exemplifies an N version programming software fault identification scheme with N=2; there were two separately designed versions of the op-

\textsuperscript{54}[22], p.3

\textsuperscript{55}IBID

\textsuperscript{56}[23], p.896
erating system. The computer system was able to identify the existence of the fault, but was unable to correct it. Thus the mission had to be delayed until the programmers could identify what the problem was and correct it.

Unfortunately, software fault tolerance does not have the history of commercial applications from which to draw positive examples. The literature contains mostly horror stories of what can go wrong without software fault tolerance.

9.1 Government Software Faults

The number of software fault tolerant systems is few. However, the military exemplifies the importance of software fault tolerance:

- The British destroyer HMS Sheffield was sunk in the Falklands war because its computer was programmed to recognize the Exocet missile fired at the ship by the Argentines as "friendly", since it is part of the British Navy's arsenal\(^{57}\). The software specifications ignored the possibility that the Exocet might be fired at the destroyer. "As a result the system ignored the transmissions of the hostile-Exocet's

\(^{57}\)[14]
homing device and allowed the missile to reach its target, namely the Sheffield."

- During the second space shuttle's operational simulation two major software faults were located. The first was a tight loop which caused the cancellation of an attempted abort. Since the abort would have only been activated in the case of another malfunction, this was a serious, life-threatening bug. The second was a bug which prevented the jetisonning of a solid rocket booster.

- A simulation of the software for the F16 showed that the navigational software caused the F16 to flip upside down when it passed the equator. The software programmer forgot about the southern hemisphere.

- The Mariner 18 aborted liftoff due to a missing "NOT" in a command.

---

58| BID
59| [25]
60| [24]. Sorry, no comments on SRBs.
61| [26], p.5
62| [27], p.1
• An F18 crashed due to a missing exception statement[^27]. Here is just one of many instances in which loss of life actually occurred.

• On Mariner 1 the Atlas booster failed during launch due to a typo.

  The statement read: "DO I=1.10" instead of "DO I=1,10"[^28].

An example of an existing military software fault tolerant system is the software for the experimental X-29 fighter. The fighter has wings which angle toward the plane's nose. This design, while permitting speed and maneuverability, is so aerodynamically unstable that three on-board computers must adjust the plane's pitch and roll 40 times a second. Each computer runs separately designed programs to execute the same functions, producing a three version software fault tolerant and TMR hardware fault tolerant computer system. If the system fails, the plane could break apart in midair[^3].

[^27]: [27]
[^28]: [28], p. 4
[^3]: [5], p. 134.
9.2 A Commercial Software Fault

While life-threatening applications are clear candidates for software, as well as hardware, fault tolerance, other applications can benefit significantly from the added reliability. A good example of this type of application occurred in late November 1985 when a software fault in a computer at the Bank of New York caused the computing system to fail, preventing the bank from closing for the day. The bank had to borrow $22.6 billion dollars from the Federal Reserve Bank. "Rumors of bank trouble triggered a run on precious metals, especially platinum, driving its price to a 29-year record." The borrowing cost the bank $5 million dollars in interest and "triggered questions from Congress and regulatory agencies." The problem was caused when an index overflowed its 15 bit length, causing progressive corruption of the data base. The program designers never anticipated such a problem. The five million dollar interest cost was only part of the expense of this error; other costs include customer inconvenience, loss of customer faith, and increased government scrutiny. Software

\[66\text{[13], p. 9}\]
\[67\text{[11], p.7}\]
fault tolerant techniques could have prevented this extremely expensive failure.

9.3 Why SFT is Not Commercially Available

Business systems normally receive ‘clean’ data. Military real-time systems, however, must contend with millions of sensor data input per second. Military computing systems must analyze the data and determine quickly what action to take. An explicit deadline exists for taking action. The flood of real-time input and the deadline requirements are not seen in commercial applications yet.

The following table indicates some of the characteristics that an application should have to profit from software fault tolerance:

- Large amounts of input
- Large amount of code
- Complicated algorithms
- Widely varying loads
- High cost of failure
• Frequent maintenance/updates

• Inability to fully test before applying

• Software failures with similar systems

• Many types of data (e.g. large # of different products)

• Many software paths infrequently taken

The Bank of New York may have prevented their costly failure had they had software fault tolerance. However, applying the above criteria to Bank of New York might not have lead us to recommend that Bank of New York use software fault tolerance. They rank somewhat highly on ‘widely varying loads’, ‘large cost of failure’, and ‘frequent maintenance/updates’, but not when compared to military real-time systems.

The commercial viability of software fault tolerance depends upon commercial systems becoming more complex according to the above characteristics. The companies to whom I spoke, even those end-users which had hardware fault tolerance and those software firms which supply turnkey systems featuring hardware fault tolerance, seemed totally complacent about
software faults. Most of the VARS, OEMs, and software houses denied that they had any problem with software bugs.

Typical was the statement of Maston Stafford, Marketing VP for Quadrastar, a Dallas-based firm supplying banking and financial software packages for Stratus hardware fault tolerant machines:

"The systems we design are fairly simple and straightforward. We have a lot of experience with similar systems. We spend lots of time and money on testing the product before it goes out the door. We never hear of problems once the product has been in place for a while."

Omri Serlin, a hardware fault tolerance consultant and editor of the industry’s newsletter, FT Systems, told me that when Tandem first started, there was little knowledge of hardware fault tolerance.

"I’m not surprised that the end-users are unaware of the benefits of software fault tolerance. But the benefits are there. You would have to educate the customers, just like Tandem did."

---

68[29]
69[30]
At another time, Mr. Serlin stated:

"I personally believe most systems, even those with hundreds of users and dozens of processes, quickly settle into predictable patterns and experience no unusual combination of circumstances, at least not frequently."

Hardware fault tolerance permitted commercial applications which never would have been undertaken without hardware fault tolerance. The key question for software fault tolerance is whether there are applications waiting to be implemented should software become much more reliable. We know of the applications in defense, but most commercial companies do not seem to have a need for software fault tolerance — yet.

---

70[31], p.12
10 Roots of HFT

Most of the improvements in hardware technology in the 1950’s and 1960’s, especially the use of solid-state devices, increased the hardware reliability of the computing system. However, replacing the magnetic core with MOS RAM actually degraded the reliability of the memory systems. With the development of error correcting code even this problem seemed 'solved'. Until Tandem's entrance in 1976 interest in fault tolerance was relegated to limited applications which required extremely reliable systems. These systems included telephone switching systems, real-time monitoring and control systems, and airline reservations systems.

The two main dimensions of hardware fault tolerance for these systems were Data Integrity (ensuring that the data entered into system is not corrupted) and Availability (ensuring that the computing system is available.) Diagram 3 illustrates a perceptual map with the positions of these early products along the relevant dimensions. These early systems included the hardware fault tolerant features required for their peculiar requirements.

\(^{71}\) most of the following is adapted from [10].
10.1 Telephone Switching Systems

AT&T sought high availability with its telephone switching system. AT&T could accept loss of data; the phone user could just redial. However, AT&T could not accept having the entire switching network crash, even for a short period of time\textsuperscript{72}. To achieve this high availability, AT&T Bell Laboratories developed its ESS computers in which all critical components were duplicated\textsuperscript{73}. One subsystem ran the telephone switching while the other subsystem remained in hot standby\textsuperscript{74} should the first subsystem fail.

10.2 Real-Time Monitoring and Control Systems

Fault tolerant applications in real-time monitoring and control systems included continuous and discrete industrial process control systems, conventional and nuclear power plant monitoring systems, aerospace telemetry and on-board control systems, air traffic control systems, and military com-

\textsuperscript{72}Its current requirement for fault tolerance is no more than two hours of downtime per forty years.

\textsuperscript{73}Its current model from this series is the 3B20D.

\textsuperscript{74}Hot standby means that the backup system actually runs jobs while available to take over for the primary computer.
mand and control systems. The systems in this group, while requiring more data integrity, still preferred high availability to data integrity.

DEC, Systems Engineering Laboratories (SEL), Data General, Interdata, IBM (IBM 9020), and NASA developed systems for some of these monitoring and control applications. An example of these types of solutions was the SEL 88, designed in 1970. The SEL 88 featured dual buses, a test bit and set instruction, dual I/O processors, parity checking, error traps, and watchdog timers\(^7^5\).

However, these early real-time monitoring and control systems contained many single points of failure (e.g. the shared memory), did not permit replacing failed components without powering down the entire system, could not be expanded beyond the two processor base model, and provided virtually no software development support.

\(^7^5\)The watchdog timer attached to CPU A would interrupt CPU B if CPU A had not completed its task by a specified time, thus guarding against complete CPU failure and infinite loops.
10.3 Airline Reservation Systems

The airlines’ needs were different from either the telephone switching systems or monitoring and control systems. Instead of requiring high availability, the airlines required database consistency and integrity. To meet the airlines’ requirements, IBM designed its Airline Control Program (ACP)\textsuperscript{76}. One of the methods ACP used to increase data integrity was a periodic dump of critical data items to a duplicate disks.

The airline reservation system was one of the first commercial on-line transaction processing (OLTP) systems. Instead of running the system in batch mode in which most computer system ran\textsuperscript{77}, the airlines required each transaction to update the database immediately. Since these systems have been installed, companies have discovered many other applications for on-line transaction processing. Gillette’s on-line inventory and distribution system using Stratus’ fault tolerant computers (described above) is a typical example of the new applications. As more important functions are placed

\textsuperscript{76}ACP was developed for the Sabre airline reservation system.

\textsuperscript{77}In batch mode updates to the main database are made at the end of the day, in a batch.
on-line, the reliability of the computing system becomes more of a concern to the company.
11 Tandem: The Pioneer HFT Company

Founded in 1974 Tandem introduced its first hardware fault tolerant series of computers, the NonStop 1, in 1976. Tandem targeted the NonStop to the on-line transaction processing (OLTP) market. All components in the NonStop are duplicated: processors, data paths, memories, disk drives, ports, controllers\(^{78}\) and power supplies. Each processor has its own copy of the operating system.

In the Tandem architecture a process runs on processor A while a copy of the process hibernates on a processor B\(^{79}\). (See Diagram 4.) Processor A sends checkpoints to the hibernating process and "I'm Alive" messages to processor B. If processor B does not receive an "I'm Alive" message, processor B's operating system assumes that processor A has failed and awakens the hibernating process B.

Other features of the Tandem system are its ability for hot repairs\(^{80}\),

\(^{78}\)Controllers provide the interface between the central processors and the peripheral devices.

\(^{79}\)"Hibernating" means that the process is not actually executing but is prepared to execute.

\(^{80}\)Hot repairs do not require powering down the entire computing system and thus losing
isolation of the processes from configuration details, and a 64K cache for each processor\textsuperscript{61}. Tandem provided a significant price/performance improvement over mainframe-based, redundant backup systems\textsuperscript{62}.

Over the last ten years Tandem has rapidly captured about 4% (roughly $600M) of the OLTP market. Tandem's success has made it one the Fortune 500 largest industrial companies (1983: \#500; 1984: \#445; 1985: \#409.) Tandem's main emphasis is in banking (26.5%), manufacturing (18.3%), communication (11.7%), transportation (6.0%), securities (5.8%), and government (3.3%)\textsuperscript{63}. It has also entered wholesale and retail distribution, POS, medical services, printing, publishing, and utilities\textsuperscript{64}. Of Tandem's 1985 revenue of $624 million, $515 million were from product sales while $109 million were from services and other revenue\textsuperscript{65}. Tandem has an esti-

\textsuperscript{81}The cache improves transaction throughput, making up for Tandem's slowness.

\textsuperscript{82}[10], p.24

\textsuperscript{83}[32], pp.2–3

\textsuperscript{84}[32], p.1

\textsuperscript{85}[3], p.20
mated 2,000 customers using about 10,000 processors\textsuperscript{86}.

Tandem has five main product offerings: NonStop 1+, NonStop II, Non-Stop TXP, NonStop EXT\textsuperscript{87}, NonStop VLX\textsuperscript{88}. The NonStop 1+ was the first fault tolerant computer, is low cost at $89K, and has limited communication capabilities. NonStop II is targeted to customers with substantial communication requirements. Its price is a hefty $155K. NonStop TXP is targeted to customers with high transaction throughput needs with two to three times the processing power of the NonStop II. It sells for $294K. NonStop EXT is targeted at the low end and was an early response to the threat presented by Stratus’ recent entry. The EXT can run without a computer room environment, offers the same power as a bare bones NonStop II, and sells for $120K, compared to Stratus’s $115 FT250. With this product Tandem targets the single small office user, for the first time straying away from its Fortune 500 clientel. Tandem just introduced the NonStop VLX in April of 1986\textsuperscript{89}. With the VLX Tandem has made its first move out of

\textsuperscript{86}Author’s deduction from 1983 Financial Report.

\textsuperscript{87}[33]

\textsuperscript{88}[34]

\textsuperscript{89}[34]
the mini-class computer and directly into IBM's central market. The VLX appears to be a response to IBM's move into Tandem's market through IBM's resale agreement with Tandem nemesis, Stratus. The VLX operates 50% faster and costs about 25% more than the TXP. More importantly is that its specifications beat IBM's.

11.1 Tandem's Susceptibility

Tandem's Strengths :

- System designed specifically for OLTP market.
- Optimized transaction integrity and throughput.
- Provision of fault tolerant networks.
- Recognized as fault tolerant leader.
- Widest fault tolerant offering.
- Easily expandable.
- Linear growth to large size has been demonstrated.

Tandem's Weaknesses :
• Operating System overhead

• Slower processing of transactions

• Application programmers responsible for hardware fault tolerance

• Lots of expensive training

• Programmer errors easy

• Hardware fault tolerance not usually implemented

• Proprietary processors

• Must program new applications from scratch

• Few existing off-the-shelf packages

• Few software development tools

• Anger from customers over high pricing

Despite the outstanding success of Tandem, several companies correctly perceived that Tandem was susceptible to attack. Tandem's susceptibility stemmed from how they implemented hardware fault tolerance and from their monopoly return on investment.
When Tandem started in 1974, Tandem decided to implement hardware fault tolerance 'quick and dirty'. Without much theoretical expertise, they decided on their interprocess communication method described above. This method which requires the processors to continuously communicate with one another required Tandem’s operating system programmer to insert a great deal of code into the operating system to implement the hardware fault tolerance, and requires the end-users’s application programmers to be continually aware of the Tandem fault tolerant scheme.

The extra overhead on the operating system required to implement this architecture slows down the processing of jobs. Not only must the processor 1) execute its user jobs, it must also 2) send "I’m Alive" messages to its shadow processor, 3) send checkpoints to the hibernating process on the shadow processor, 4) receive "I’m Alive" messages from the processor it is shadowing, and 5) receive checkpoints from the active process on the processor it is shadowing. (See Diagram 4.) While Tandem has attempted to correct these problems by adding fast buses and caches, the slowdown resulting from the operating system overhead is intrinsic to Tandem’s approach.
That end-users's application programmers must be aware of the Tandem architecture and fault tolerant scheme and must program checkpoints explicitly into the application program, has lead customers to voice two central complaints:

1. Application programmers must go through extensive training, spread out over a half year, at Tandem facilities, just to be able to apply the Tandem computer to their application. In addition, each new programmer the customer hires must take the Tandem courses, providing a continual expense for the customer. The resulting application software is specific to the Tandem architecture; it cannot be ported over to other machines. Finally, software developers cannot use off-the-shelf software or tools which were developed for other computer manufacturers' architecture.

2. Most real Tandem systems do not use the Tandem in fault tolerant mode (called "Duplex Mode"). Because duplex mode requires additional programming effort, skill, time, and money, over 70% of the programs running on Tandem computers do not run in duplex
mode\textsuperscript{90}. Tandem has attempted to automate much of the checkpoint assignments, but special programmer attention is still required.

Many customers were disturbed with the amount of training required, the slow processing time, and the disappearance of the fault tolerant benefits with real systems. The West Lynn Cremery, the first commercial OLTP customer to purchase a fault tolerant computing system from a fault tolerant company beside Tandem, describes their problem:

"We sent the programmers out to (Tandem’s) training center in Chicago ... After two weeks the programmers were fed up. It was just too difficult to develop a fault tolerant system on Tandem\textsuperscript{91}."

Another problem with Tandem's architecture is that it identifies hard faults (which cause the subsystem to fail) but not soft faults (which causes the subsystem only to err, but not fail.) For example, as long as CPU A in Diagram 4 keeps sending "I'm Alive" messages, CPU B will not interfere in CPU A's processing of the job. However, CPU A could be faulting, outputing garbage.

\textsuperscript{90}[35]
\textsuperscript{91}[36]
Tandem’s behavior while holding a monopoly position in the industry led to further susceptibility. Tandem kept its prices very high — for its hardware products, its software products, its service, and its training. These prices angered Tandem’s customers, especially when the aftersale expense of the Tandem remained high. Customers who wanted hardware fault tolerant features but could not afford Tandem’s high prices, were happy to welcome new competitors. Tandem’s high ROI also attracted a lot of attention from venture capitalists and from more technically proficient engineers.
12 Enter the Startups

Starting in 1980, at least twelve hardware fault tolerance startups received venture capital funding. By 1986, all but three of these firms were doing poorly, seeking an acquirer, or shutting down their doors. Two that did well (Triconex and Stratus) both made substantial joint marketing agreements with major traditional manufacturers (Honeywell and IBM respectively).

Besides the twelve explicitly hardware fault tolerant startups, other startup companies, like EnMasse Computers, received venture capital funding promising to include hardware fault tolerance in their product but when the hardware fault tolerant feature proved to be too difficult to implement, they dropped the hardware fault tolerance feature.

12.1 Industrial Automation Startups

August Systems, Inc

(Tigard, OR)

Autech Corp
Of the twelve explicitly hardware fault tolerant startups, four companies have not targeted the On-Line Transaction Processing (OLTP) market, choosing instead to focus on the industrial automation and process control markets. August and Triconex have both done relatively well, although Triconex is in better shape than August. Interestingly, both of these startups are the only two to have implemented the theoretically sound method of hardware fault tolerance – TMR.

August developed a system based on Intel Corporation's 8086 microprocessors\textsuperscript{92}. Each of the three 8086's run the identical code, checking their\textsuperscript{92} the same ones used in IBM PCs.
output against each other in true TMR fashion. August holds 10% of the Industrial Automation market compared to 83% for Tandem and 7% for Stratus\textsuperscript{93}.

Triconex (Irving, CA) was begun in Sept. 1983 by an officer of August Systems. Triconex supplies TMR programmable logic controllers (PLCs) to continuous process manufacturers.

Process control, for which PLCs are used, is divided into continuous and discrete parts control\textsuperscript{94}. Discrete parts control applies to assembly lines handling individual, discrete parts. For example, an auto plant producing fenders is producing discrete parts. Continuous control applies to manufacturing processes involving the flows of fluids. For example a paint company would have continuous controllers to control the flow of pigments, chemicals, and water that compose the paint. While the larger market segment is the discrete parts, the greater need for the fault tolerant feature lies with the continuous process control applications where a faulty controller might allow thousands of dollars of chemicals to contaminate the entire

\textsuperscript{93}[37], p.3

\textsuperscript{94}[38]
days batch of product.

In January 1986\textsuperscript{95}, Triconex entered into an agreement with Honeywell, permitting Honeywell to resell Triconex computers\textsuperscript{96}. The Triconex price is high for PLCs ($65K) but their utility for certain highly critical tasks is worth the price. A continuous flow engineer confided that he would gladly pay three times the price for a fault tolerant PLC, but that the PLC had to be made by a major manufacturer\textsuperscript{97}. This reflects the inherent conservatism in this market; few engineers are willing to risk their jobs on an unproven startup. Triconex's joint venture with Honeywell permits them to have that required name behind them. Interestingly, Triconex has been able to raise $6.7M in three rounds of financing\textsuperscript{98} despite the recent demise of many of the fault tolerant computer companies and the slower-than-expected growth in the PLC market.

Both Autech and H-P are planning to offer minicomputer systems for

\textsuperscript{95}[11], p.6
\textsuperscript{96}The industry term for Honeywell in this type of agreement is Value Added Reseller, or VAR.
\textsuperscript{97}[50]
\textsuperscript{98}the last during Feb, 1985
industrial automation — dual systems with one the systems in hot backup.

A watchdog timer ensures that both processors continue to run. Should one processor not complete its task before the timer runs out, the shadow processor takes over the job. Hewlett-Packard has run into severe technical problems getting their system out the door\textsuperscript{99}, and it is questionable whether fault tolerance will be a part of that product if it ever does arrive.

12.2 On-Line Trasaction Processing Startups

Aragen Systems Corp

(Fair Lawn, NJ)

Computer Consoles, Inc

(Rochester, NY)

NoHalt

(Farmingdale, NY).

\textsuperscript{99}[47],p.10
Parallel Computers, Inc
(Santa Cruz, CA).

Sequoia Computers
(Marlboro, MA).

Senapse Computer Corp.
(Milpitas, CA).

Stratus Computer
(Marlboro, MA).

Tolerant Transaction Systems
(San Jose, CA).

Auragen Systems (Fair Lawn, NJ) which started in 1980\textsuperscript{100} ceased op-\hfill

\textsuperscript{100}[10], p.26
erating in late June 1985\textsuperscript{101} after spending $21 million of venture capital. Auragen attempted to implement a multi-processor and fault tolerant system using Motorola 68010 microprocessors. Three 68010 processors formed what Auragen called a "cluster". Two of these processors would execute user processors. This is similar to Tandem's approach, in which two processors execute different processes, but each keeps a hibernating copy of the other processor's processes. Auragen used "I'm Alive" messages to inform each processor that the other processor was still functioning. The third processor manages the system's bus and executes the operating system kernal\textsuperscript{102} functions\textsuperscript{103}.

Auragen's approach differed architecturally from Tandem's in that both the active and hibernating processes received and saved all input, instead of having the active process checkpoint to the hibernating duplicate process. Periodically the active process would update the hibernating process. Should the active processor fail, the hibernating process would take over the

\textsuperscript{101}[31], p.14

\textsuperscript{102}The kernal contains all the core functions of the operating system.

\textsuperscript{103}What if the third processor fails? No fault tolerance there!
user task, processing all input and suppressing all output until it reaches the point in the active process where its processor failed. This approach transfers the responsibility for inserting checkpoints from the application programmers to the operating system, thus eliminating one of the key drawbacks to the Tandem system.

Auragen's difficulties stem from trying to implement a rather complicated fault tolerant strategy in conjunction with a multiprocessing environment. For an operating system, Auragen decided to modify Unix to both the multiprocessing and fault tolerant environment. This modification of Unix proved to be fairly difficult because Unix is designed neither for fault tolerance or multiple processors\textsuperscript{104}. Because of these technical problems, Auragen never produced a fault tolerant computer. Nixdorf Computers purchased some fifty non-fault tolerant single clusters from Auragen. When Auragen could not raise further funds, Nixdorf announced it would introduce its own Auragen-based system\textsuperscript{105}.

The problem with adapting Unix to a fault tolerant environment also

\textsuperscript{104}[46], p.83

\textsuperscript{105}[45], p.14
plagued three other firms: Tolerant, Sequoia, and Parallel\textsuperscript{106}. Tolerant Systems (San Jose, CA) employs two National Semiconductor Coporation 16000 microprocessors in a synchronization scheme similar to Auragen’s scheme. It has recently announced an upgraded system using the faster NSC 32000 selling for $190K\textsuperscript{107}. Tolerant has actually shipped its system, although attempts to raise another round of venture funding in June 1985 failed. Tolerant has barely survived through a joint venture with a European and a Japanese concern. These two firms have taken an equity position in Tolerant, providing Tolerant with working capital. With a version of their system running ADA\textsuperscript{108}, Tolerant hopes to focus on the government systems.

Sequoia Systems (Marlboro, MA) has not fared as well as Tolerant. The company, founded in September 1981, has had severe difficulty adapting Unix. It finally had to replace Unix’s kernal with its own proprietary version. Sequoia sought to develop a multiprocessing system in which the

\begin{footnotesize}
\textsuperscript{106}[46],p.82
\textsuperscript{107}[53],p.12
\textsuperscript{108}ADA is a procedural programming language which the defense department commissioned. All software written for the defense department must be written in ADA.
\end{footnotesize}
processors check themselves. Should one identify a problem, it automatically disconnects itself\textsuperscript{109}. The next available processor takes over the processing from where the disconnected processor left off.

After first introducing its product in 1984 after severe technical delays resulted in a top management purge, Sequoia had to recall its product. Sequoia was barely limping along on bridge loans\textsuperscript{110} while it was trying to develop an OEM arrangement with Sperry for its newest product. In December, Sperry, which had expected a faster 68020-based system from Sequoia, ended VAR talks, placing the future of Sequoia in severe doubt.

Parallel Computer (Santa Cruz, CA) is attempting to raise another round of financing\textsuperscript{111} to complete its Unix-based operating system. Parallel offers a product at the low end of power and price, uses two 68010's with a hibernating processes and synchronization (similar to Auragen's doomed architecture). Parallel claims to have a $5M order backlog\textsuperscript{112}, but the firm appears near to death.

\textsuperscript{109} A question might be what if the auto disconnect fails.

\textsuperscript{110}[11], pp.1-2

\textsuperscript{111}[13], p.8

\textsuperscript{112}[53], p.10
Synapse Computer, founded in 1980, shipped its first system in 1983. The company also had to recall its product and redesign it. It reshipped its newer product in 1984, while beginning to look for a buyer. The delays and disappointment cost the top management their jobs. Synapse implemented an interesting 68000-based architecture it called “N+1”. To implement fault tolerance, the architecture requires only one more processor than the N required\textsuperscript{113} to execute the system jobs in a non-fault tolerant computer. Should a processor fail, the next available processor takes over the job which has been checkpointed to a common main memory. The Synapse approach did not require programmers to checkpoint explicitly. Insertion of checkpoints was taken care of by the compiler. The Synapse system also permitted a graceful linear growth.

Synapse raised the most venture capital of any of the fault tolerant OLTP companies — $35M in five investment rounds\textsuperscript{114} but shipped only a few systems. The main problem appeared to be implementing its proprietary operating system. In May, Synapse was about to be acquired

\textsuperscript{113}Versus the 3 * N required for TMR.

\textsuperscript{114}[35], pp. 1–2
by McDonald Douglas Computer Systems (MDCS). Upon closer scrutiny, MDCS declined to go through with the deal and Synapse closed its doors.

12.3 Reasons for the Poor Performance

What is remarkable is how poorly these startups have fared.

The startups ignored the theoretically sound TMR approach and implemented a cumbersome, but proprietary, hardware fault tolerant scheme. This led to product delays when their clever design proved unworkable. Omri Serlin notes\textsuperscript{115}:

\begin{quote}
The current crop of fault tolerant system suppliers have been somewhat naive in their estimation of how quickly they can develop the clever architectures they envisioned.
\end{quote}

Almost all of the firms ignored the difficulty of implementing the software. Perhaps this is due to the traditional dichotomy between hardware and software engineers; most of these firms were begun by hardware engineers. However, while their hardware problems were solvable, their software

\textsuperscript{115}[51]
problems were not. Whether the firm developed their own operating system or adapted Unix, they failed.

Perhaps the cornerstone to their error was the belief that they could establish new standards for hardware and software in an industry dominated by IBM and DEC. End users were concerned about their current hardware investment and software houses were concerned about retaining their investment in their software tools, training, and packages. Bringing out an entirely new machine required these customers to throw away their investments. A software house, Ann Arbor Company, reported that\textsuperscript{116}:

\begin{quote}
Everybody is trying to standardize on hardware. More often than not, we’re told what computers to use.
\end{quote}

The missed milestones come directly from the software difficulties. The extra expense to develop the product drained all the venture money from the companies. Little was left for marketing the product. Venture capitalists, not known for their patience, were increasingly quick to pull the plug when the public market went sour in 1983, especially when the companies

\textsuperscript{116}[37]
came to them without their milestones, without their product, and without sales

No management came from Tandem. The managers all had to learn on the job. Many officers did not survive; top management from Auran, Synapse, CCI, and Sequoia were fired. Interestingly, Triconex had officers who were weaned on August; they were able to apply their lessons to Triconex.

The companies were extremely confused over who their customers were and what those customers’ needs were. The companies wanted the customers’ needs to be fault tolerance. But even a superficial examination of the customers proves that customers have much more complicated needs than just that single feature. What they needed were business solutions. In addition the hardware fault tolerant startups ignored the central role of the OEM and their requirement for programmability and tools. Even Tandem ignored the turnkey, OEM, and VAR market, preferring to use its large salesforce. Only 5% of Tandem’s sales were not due to its salesforce.
13 Stratus: A Successful Startup

The only 1980’s startup to succeed dramatically in the fault tolerant OLTP market is Stratus Computers (Marlboro, MA). Stratus developed its product on time and delivered it with enough marketing dollars to actually sell it.

While Tandem had captured the entire fault tolerant market, it had kept its systems high priced and high end. With limited manufacturing capacity and a monopoly position, Tandem did not want to serve the low end. Stratus aimed its product to that low end. Stratus’s highest priced product is $300K versus a high end Tandem system of $1 million.

Stratus’s first customer was West Lynn Cremery (Weston, MA) in 1982. West Lynn\textsuperscript{117} needed a replacement for its Nixdorf 600 computer which West Lynn had outgrown the system, was tired of its frequent downtime, and was annoyed with Nixdorf’s poor service. West Lynn talked to Prime, Data General, Sperry, and finally Tandem, requesting proposals for a new system. Tandem convinced them of their need for the fault tolerant feature.

\textsuperscript{117}[36]
West Lynn asked their main programming firm, Femcon (Weston, MA), to adapt West Lynn's current telemarketing and truck loading system to Tandem's system. Two programmers flew to Chicago to start Tandem's training program to learn how to program on a Tandem. The programmers were horrified to find how difficult the Tandem architecture was to program. Tandem's inflexibility regarding discounts further prompted West Lynn to expand their search.

The manager of West Lynn's information services had read an article in Computer about the new Stratus offering. West Lynn and Femcon asked for a demonstration. Femcon was so excited about how easily the Stratus system was to program that they convinced West Lynn to continue examining Startus.

Stratus made an intensive personal sell to West Lynn. Top officers blocked out how the system would run on Stratus, compared the benefits of Stratus's hardware-based hardware fault tolerant system to Tandem's software based hardware fault tolerant system, stressed the ease of developing and maintaining the software, and noted the price which, while slightly
higher than Tandem’s initial price, was competitive with both Tandem’s\textsuperscript{118} and the traditional manufacturers’ systems.

Tandem, confident that it had the West Lynn account, ignored West Lynn, leaving them alone to examine Stratus’s offering in detail. When West Lynn decided on Stratus, shocked Tandem officers flew to West Lynn to find out about the new competition.

West Lynn purchased only a single FT200. It has upgraded to two XA400’s and currently has seventy-five terminals attached to the system. The company claims to have experienced no downtime in the three years and does not keep an operator in the computer room. (Quadstar\textsuperscript{119} estimates that roughly half of its customers do not have a computer room operator.)

West Lynn is particularly excited about Stratus’s service. Should a module go down, the Stratus computer automatically dials Stratus and requests the replacement. West Lynn rarely knows about the hardware problem until a replacement arrives in the mail.

\textsuperscript{118} Stratus’s price is lower when adding Tandem’s hidden costs.

\textsuperscript{119}[29]
While Tandem is the market leader in the fault tolerant industry, Stratus has quickly captured 10% of the fault tolerant OLTP market. Stratus has major resale agreements with Olivetti (who sold 75% of Stratus’s foreign computers in 1984), Burroughs, Honeywell, and IBM\textsuperscript{120}.

13.1 Stratus’s Architecture

Stratus uses off-the-shelf processors (Motorola 68000’s) in an architecture it calls "pair and spare". (See Diagram 5). Stratus’s architecture uses four processors, which together Stratus calls a “processing module”, to implement fault tolerance. The four processors are broken into two pairs. All processors run an identical copy of the user task. Each processor compares its output with the output of its companion. Should a comparison fail, the pair of processors which failed the comparison pull out, and the other pair (the ‘spare’) continues to process the user task, uninterrupted. Stratus requires no recovery from a fault — thus no checkpointing or “I’m Alive” messages (as opposed to Tandem). Also, little special programming attention is required to implement fault tolerance on Stratus computers (as

\textsuperscript{120}[49]
opposed to Tandem computers).

Stratus based its operating system on Unix but optimized it for transaction processing. Interestingly, Stratus was one of the few startups to recognize the key role the operating system plays in getting the system to market. Stratus brought on some of the best people they could find to write their operating system. Stratus has a significant presence in banking and finance applications.

Stratus's Strengths:

- Hardware fault tolerance implemented in hardware.
- Fault tolerance is transparent to application programmer and to user
- Repairs are autodialed to Stratus service center
- No recovery required if fault occurs — other pair simply keeps processing.
- No checkpointing or "I'm Alive" signal — faster
- High redundancy for low price
- Hot repairs and hot expansion
• Easy to program

Stratus's Weaknesses:

• Must build own circuit boards
• Limited expandability (32 processing modules maximum)\textsuperscript{121}
• Must add entire processing module to expand
• Difficult to share resources among processing modules.

13.2 CCI: Another Success?

Computer Consoles (Rochester, NY) is another rather successful hardware fault tolerant computer, although they have just replaced almost their entire top management with some very impressive people (including ex-Wang John Cunningham, and ex-GTE Vanderslice) Computer Consoles main presence is in directory assistance. They have optimized their system for querying (as opposed to updating). They replicate the data base among several disks increasing the speed with which a search can be conducted.

\textsuperscript{121}This maximum level is frequently questioned and has not been verified by an outside source.
Since the major time delay is in disk access, Computer Consoles is able to show significant speed advantage. In an interview with the ex-VP of Software Development\textsuperscript{122} for CCI, I was told that CCI can provide (for $1.2 million) a system which runs 4000 transactions per second (compared to Tandem and Stratus systems running about three to thirteen TPS.) CCI sold $131.2 million worth of products in 1984\textsuperscript{123}, making them the second largest fault tolerant manufacturer. However, they do not directly compete with either Stratus or Tandem.

13.3 Lessons from Startups

We can draw some valuable lessons from the startups to guide future ventures:

- Fault tolerance is a nice feature for the customer, but is not the only feature.

- The end-user wants a hassle-free solution to her business problem.

\textsuperscript{122}[52]
\textsuperscript{123}[37]
• The customer is greatly influenced by the software house designing the system.

• The software house seeks a system that is easy to program.

• Nominal price, as long as somewhat comparable, is not a major consideration. Long term cost considerations play an important part.

• No headache maintenance is important.

• Most purchases are due to dedicated, intensive sales efforts.

The importance of the software house cannot be overemphasized. In the West Lynn example, had the software house not complained so much, West Lynn would be using a Tandem system today. A vice-president of Femcon stated\(^{124}\): "Tandem required a lot of programming overhead. This additional programming increases the final cost of the system to the customer."

Femcon is one the the most experienced Stratus software houses. Femcon and Stratus frequently cooperate on sales. Sometimes Stratus will

\(^{124}[35]\)
introduce the end-user to Femcon; other times Femcon will ask Stratus to bid on a user system with them; and sometimes they will combine to target certain customers.
14 Role of VAR

The major trend in the hardware fault tolerant industry is the value added reseller (VAR) agreements between the fault tolerant startups and the traditional manufacturers. The benefits to the startup are:

- a penetrated customer base
- an experienced sales force
- additional managerial guidance
- increased compatibility
- a big name behind the sales.

As Omri Serlin says\(^{125}\):

"The potential for fault tolerance is large. The principle reasons for (Stratus) not selling more (fault tolerant computers) is compatibility and a big name."

The benefits to the end-user are:

\(^{125}[39]\)
• assurance of viability

• purchase from name brand

• assurance of future compatibility

• increased support from OEMs and software houses.

The traditional computer manufacturer is able to:

• reduce R&D expense

• examine market closer

• examine possible acquisition candidate

• choose a proven fault tolerance implementation

Even the hardware fault tolerant startups that failed, did not do so until after remarketing agreement possibilities were exhausted:

• Auragen closed its doors after Nixdorf refused to resell Auragen.

   Auragen finally sold its designs to Nixdorf.

112
• Synapse counted on a joint venture with McDonnell Douglas’ MDCS. When that fell through, they sought to be acquired by MDCS. When MDCS declined, Synapse went out of business.

• Sequoia sought to develop an VAR arrangement with Sperry. When Sperry dropped discussions, Sequoia all but left the market.

• Tolerant has survived only because of a joint venture with a European and a Japanese concern.

The hardware fault tolerant startups which succeeded did so with the help of the joint ventures:

• Triconex joint markets with Honeywell

• Stratus joint markets with Olivetti, Burroughs, Honeywell, and IBM.

14.1 Stratus’s Lead with the Vars

Stratus is the most successful company to use these resale agreements to leverage itself into the market. Stratus’s President Foster spent his time looking for new sales channels for his company’s products. While Tan-
dem kept its sales mostly in-house\textsuperscript{126}, Stratus had neither the established, large salesforce, nor the high-margined product\textsuperscript{127} to compete with Tandem directly. Omri Serlin explained\textsuperscript{128}:

"There was no way a small company could effectively cover all the angles in this market and achieve satisfactory growth through a direct sales force alone."

Canin, a senior technology analyst for Hambrecht \& Quist\textsuperscript{129}, agrees\textsuperscript{130}:

"They have everything in place to become a major company."

While Stratus's joint resale agreement with System Development Corporation, a Burrough's subsidiary, is estimated to be worth $50 million to Stratus over the next five years\textsuperscript{131}, the deal that really turned the heads of the OLTP customers was the January 1985 resale agreement with IBM.

\textsuperscript{126}some 95\% of 1983 sales were initiated by Tandem's large sales force

\textsuperscript{127}Since Stratus was targeting the low-end of the market

\textsuperscript{128}[40] p.1

\textsuperscript{129}a venture capital firm which has invested in a number of fault tolerant startups, including Stratus

\textsuperscript{130}[40]

\textsuperscript{131}[41] p.1
Foster reportedly spent three years courting IBM until IBM finally agreed. Estimates of the worth of the agreement are purely guesswork, mostly because most analysts are unsure of IBM’s motive for the agreement. As Omri Serlin says:\textsuperscript{132}:

“There is a consensus within IBM that the fault tolerant market is not worth enough or long-lasting enough to justify committing resources to a fault tolerant system of its own.”

IBM’s ambivalence was noticed when IBM released the Stratus machine without making it more compatible with its own IBM machine. Mr. Greco, VP of Financial Services for Interactive Data Corp, said:\textsuperscript{133}:

“It’s not an IBM operating system. Basically a customer would have to start from scratch (to design a system on the IBM Stratus). If a customer installs the system now, they are going to have to design their own applications to run on their system. I don’t know how many people are going to take this

\textsuperscript{132}[2]

\textsuperscript{133}[44]
machine now because its not even compatible with IBM systems yet."

Many analysts feel that IBM tried to develop its own fault tolerant computer but was unable to do so. Serlin doubts that IBM is still trying to produce its own fault tolerant computer:\(^{134}\):

"IBM’s user base is too large. The architecture of its main-frame is too limited in its range of design options to allow fault tolerance. IBM would not want to cannabalize its existing customer base. (The Stratus agreement) gives IBM a product for their customers who say, ‘I’m just not going to buy anything but a fault tolerant computer.’ That was a situation that Tandem and Stratus took business away from IBM."

IBM’s move validates the fault tolerant industry. Fielder, a software house for Stratus, described the about-face that IBM had to make to agree:\(^{135}\):

\(^{134}\)[42]

\(^{135}\)[43]
“Until now IBM has been telling its accounts that they didn’t need fault tolerance. They could link together large IBM mainframes to get the fault tolerance. The customers would have to pay twice the amount just to have one IBM machine stand idly by.”

While the IBM deal appears to be a blessing, there is some concern whether Stratus will be able to retain its independence should the deal work out well. David Wu, an analysts for Montgomery Securities, said that Stratus should become concerned if IBM’s demand for Stratus’s computers became so great that Stratus’s manufacturing capacity became saturated. Stratus would then become a captive OEM of IBM\textsuperscript{136}.

Stratus could also end up like ROLM; speculation is that IBM chose Stratus instead of Tandem, not just because Stratus’s approach to fault tolerance is better, but because a Stratus acquisition would be less of a problem if IBM were acquiring the number two hardware fault tolerant manufacturer.

\textsuperscript{136}\textsuperscript{40}
15 Conclusion

What are the conclusions we can draw for both the hardware fault tolerant and software fault tolerant markets?

15.1 HFT Conclusions

With the major thrust of the traditional manufacturers into the highly lucrative hardware fault tolerant niche, we can assume that the entrepreneurial period for the hardware fault tolerant industry has ended. Those startups which are struggling along now will find it increasingly difficult to sell their systems independently of the traditional manufacturers. A new venture in this market will require major funding and a major name behind it. Without the large salesforce, the assurance of viability, the compatibility, and the large number of turnkey houses and software houses behind a hardware fault tolerant newcomer, that newcomer must fail.

The traditional manufacturers will quickly absorb the hardware fault tolerant subindustry. IBM will acquire a significant portion of Stratus, or come out with its look-alike system. H-P, NCR, Honeywell, and Sperry are all busy working on their in-house hardware fault tolerant computer. Even
Tandem is facing a bleak future without major backing. That backing will not be forthcoming. Tandem is tied to an antiquated fault tolerant scheme; it cannot compete effectively with the newer approaches to fault tolerance. The newer hardware fault tolerant computers to be offered by the traditional manufacturers should be able to leapfrog the Tandem offering easily.

The user will drive this demise of the hardware fault tolerant industry. The user wants a total business solution, better offered by the more integrated traditional computer companies. Fault tolerance has always been just a single feature, important to many users, but not important enough for them to ignore the other shortcomings of the hardware fault tolerant startups. To the user, compatibility of the new system with their existing hardware, availability of software packages, and minimization of personnel training are extremely important issues. While many customers will purchase the single feature package, these customers will probably not stay loyal if traditional manufacturers offer a similar package. The lesson here is that a startup must be aware of how important compatibility has become. Instead of developing an entirely new system, the startup should focus on
trying to integrate many of the benefits the user already receives with the single feature the startup is providing.

The major role of the OEM and software house in the buying decision is another lesson. Tandem tried to bypass these intermediaries with a direct sales force. When Stratus offered them an easily programmable alternative, and were interested in using them as a major sales channel, many of the software houses converted to Stratus. The needs of these intermediaries are significantly different from those of the end-user. They want to be able to get the system up as quickly as possible, using as many tools as they can to make their job easier and more profitable. Without the turnkey houses, other startups were doomed from the beginning. They found themselves in the position of selling a hunk of hardware, and not a business solution.

Hardware fault tolerance is the wrong focus for a fault tolerant company. Most of the failures (50%) occur from software, not hardware faults. Another third occur from operator errors. The hardware fault tolerant companies offered a solution which, at its best, would solve only 20% of a customer’s downtime. In return for this minimal increase in uptime, they forced their customers to write new software (thus increasing the like-
lyhood of software faults) and to adopt new operating procedures (thus increasing the likelihood of operator faults). Software fault tolerance is a harder problem; this might be why none of the companies have addressed it. But even in hardware fault tolerance, the area where these companies were supposed to be strong, the companies implemented the hardware fault tolerance in such a way as to reduce much of the added benefit. If a large body of theoretical knowledge and lead-user examples exists regarding how to implement a feature, why choose a more difficult, and less successful, approach?

The surprising fact is that Tandem made such large profits, and that Stratus will do extremely well, despite their technical and marketing errors. Certainly, there are the large numbers of startups which had to pay the price of their errors. However, much money was made in this temporary industry. This illustrates how much room for error there is when a company can identify a high need user group. The fault tolerant companies had to increase the users’ awareness of the benefits of fault tolerance, and wait until more systems came on-line which required a high degree of uptime. But they found their niche and rode the trend into the ranks of the large
companies.

15.2 SFT Conclusions

What lessons are applicable for a software fault tolerant industry?

While lead users are clearly identifiable for software fault tolerance, commercial applications will not arise in the near-term. The Department of Defense is intensively pushing the state of the art in software fault tolerance as they push the state of the art in real-time systems. The possibility exists that a software fault tolerant startup which focuses on Defense applications may be able to design workable, proven systems with government money. However, the normal difficulty exists of how to transfer that technology from life-destroying to life-enhancing applications. DOD will not permit commercialization of major breakthroughs, and a company raised by government contracts is seldomly weaned from the taxpayers’ teat.

Most commercial applications are still too basic to receive the full benefits of software fault tolerance, although there are extremely large, complex systems which might benefit from software fault tolerance. The question arises whether managers are aware of software fault tolerance. Most man-
agers I talked to were unaware of software fault tolerance. While they
did have an occasional software fault, their systems frequently did not ex-
hibit the required behavior (suggested above) for systems to benefit from
software fault tolerance. As more critical, large, and complex systems are
implemented, and as hardware fault tolerance decreases the more obvi-
ous failures, awareness and concern over both software and operator faults
will increase. Tandem's role in educating the users about the benefits of
hardware fault tolerance should not be forgotten. Without that education,
the higher demand for hardware fault tolerance might not have occurred so
quickly.

Even if customers were clamoring for software fault tolerance, software
fault tolerant theory remains in its infancy. Significant theoretical questions
concerning the independence of design fault assumption prohibit academics
from being able to recommend a software fault tolerant technique. Another
question regarding the independence of specifications is also unanswered.
Finally, the practical question of how to implement and how to defray the
costs of implementing software fault tolerance remains

But drawing from the hardware fault tolerance example, one would
wonder if these questions need to be answered. Tandem entered the market with a theoretically unsound idea. Even Stratus's design has single points of failure and requires too much hardware. These companies never bothered to optimize their design; they sought only to satisfice. And the end-user was fairly happy with the result. If these companies could make fantastic profits while ignoring theory, why can't a software fault tolerant company take the same tack?

We know from the hardware fault tolerant example, that the first to address the problem, once the problem begins to arise in the market, should be able to achieve a very large ROI. Perhaps it would be best to come in with a mediocre solution, simply to get into the market quickly. On the other hand, Tandem's poor design attracted quite a few people who felt that they could beat Tandem.

Software fault tolerance benefit end-users. It can decrease system downtime almost by half. It can permit the user to increase her reliance on automated systems. And it has the potential for defraying its costs through less programming concern over fault recovery, less testing and assurance, and increased revenues from higher uptime. Since the 'best' answer to software
faults so far is three version programming, a startup could combine the benefits of both a hardware TMR approach with the software fault tolerance. If the system also was familiar to users (or, better, if the system had an artificial intelligence interface), the system could even solve operator faults. The market for a system which assures error-free running of jobs would be great.

Another lesson we can bring directly from the hardware fault tolerant companies is not to ignore the role of the turnkey houses and the software houses. Since N version programming requires coordination of several software houses, and since these houses are usually in competition with one another, a software fault tolerant company would have to pay much more attention to this link in the chain. Since a software fault tolerant company would not provide business solutions, but instead, an environment in which the turnkey houses could provide software fault tolerant business solutions, the OEM, software house, VAR channel would be all important.

What could the software fault tolerant company expect in the future? If the software fault tolerant company is able to achieve high ROI, the firm can expect many startups to come nipping at its heels. Tandem ignored
its competition from Stratus until it was well-entrenched. A software fault tolerant startup would have to keep track of potential new ventures, be able to price aggressively, and be able to change its basic software fault tolerant approach. The last item is by far the hardest; Tandem has so many customers currently dependent upon its older design that it cannot change over.

The software fault tolerant startup can also expect the traditional manufacturers to come into its market — even more quickly than they did for the hardware fault tolerant compies since they have done it once before. The best defense here would be not to fight the traditional manufacturers, but to try to joint venture with them as much as possible. Both can profit from the venture.
16 Appendix


[4] Interview with Dedra Casy, System Manager, Gillette, Andover, MA.


[7] Interview with John Rollwagen, President, Cray Computers, MN.


128


[29] Interview with Maston Stafford, VP, Marketing, Quadstar, Dallas, Texas.

[30] Interview with Omri Serlin, President ITOM, Los Altos, CA.


[34 ] "Tandem Computer Expected to Introduce Transaction Processor", 

[35 ] Interview with Ed Shrotto, VP Business Relations, Femcon, Weston, 
     MA.

[36 ] Interview with Jack O'Neil, Manager of Information Services, West 
     Lynn Cremery, Inc., Weston, MA.


[38 ] Interview with Peter Bartlett, Executive Vice President, Automation 
     Systems Inc., Eldridge, IA.


[40 ] Robert Hertzberg, "Stratus reaping benefits of new sales channels", 


[50] Interview with Joseph Andel, Design Engineer, DuPont, NJ.


132
[52] Interview with Tabloski, VP, Software Development, Thinking Machines, Inc, Cambridge, MA.

Diagram 1:
N Modular Redundancy
N = 2
Diagram 2: N Modular Redundancy
N = 3
Diagram 3:
Trade-Off of Early Systems
Diagram 4:
Tandem's Architecture
Diagram 5:
Stratus's Architecture