A Software Implemented Fault-Tolerance Layer for Reliable Computing on Massively Parallel Computers and Distributed Computing Systems

Gavin D. Holland   Dhiraj K. Pradhan

Department of Computer Science
Texas A&M University
College Station, TX 77843
(409) 862-2438
E-mail: [gholland, pradhan]@cs.tamu.edu

Abstract

A novel architecture for a software-implemented fault-tolerance layer for application reliability on massively parallel computers and distributed computing systems is proposed. This is the first attempt at providing a purely software-based, user-level solution for fault detection, reconfiguration, and recovery in a parallel environment. The symmetrically distributed, multi-tiered layer envelopes user applications, enabling it to perform fault-tolerance related actions apart from, and transparent to the application. It's modular design enables dynamic runtime selection of the most appropriate fault-tolerant algorithm, and is, therefore, not restricted to one particular fault-tolerant method. Performance and coverage measurements of a minimal implementation of the proposed layer are presented, and indicate that user-level software-implemented fault-tolerance is realistically doable and reasonably efficient and effective.

Key Words: Software Implemented Fault-Tolerance, Massively Parallel Processing
Word Count: 5,425

*This work was supported by NSF under grant #DMI-9361288

1 Introduction

It is widely accepted that massively parallel processors (MPPs) will eventually replace vector supercomputers as the platform of choice for commercial and scientific applications. In fact, traditional vector supercomputer vendors, like Cray and Convex, have signalled this end by introducing their own line of MPPs, the T3-D [2] and Exemplar [1].

Most of the current MPPs are designed around the distributed-memory, MIMD architecture. This architecture is characterized by multiple microcomputers, each with its own microprocessor and memory, interconnected by a high-speed network. Unlike the SIMD architecture, each microcomputer, or *node*, is capable of independent execution, and is typically based around a commodity microprocessor. For example, Intel uses i860 XP microprocessors in its Paragon XP/S [10], IBM uses RS/6000's in its SP2 [8], Cray uses DEC Alpha's in its T3-D [2], and Meiko uses 90MHz SPARC's in its CS-2HA [13]. Although some vendors, like nCUBE [14] and Parsytec [18], use proprietary microprocessors or transputers which may incorporate special features, most vendors rely on quantities of inexpensive, commodity microprocessors for their computing power.

Unfortunately for all, larger numbers of microprocessors mean drastic reductions in overall reliability. It is a common misconception that MPPs are naturally reliable, fault-tolerant, and reconfigurable because of their abundance of resources. In fact, the only attribute gained by multiple processors is a higher total failure rate [12]. Intuitively, as MPPs grow to thousands of microprocessors, the failure rate can also be expected to grow thousands of times more than a single microprocessor system. Only a few vendors, such as Tandem [6], Sequoia [16], and Stratus [19], have devoted serious attention to this problem, providing ultra-reliable MPPs that utilize combined hardware- and software-implemented fault-tolerant techniques [5]. However, more MPP vendors are now realizing that the scientific and commercial users are as concerned about high-reliability and high-availability as high-performance.

To bolster reliability, vendors such as nCUBE, Intel, Parsytec, and Meiko are employing hardware-implemented fault tolerance techniques, such as coding in memory and busses and redundancy in the interconnection network. Although hardware-implemented techniques are transparent, efficient, and effective, techniques beyond the use of codes are difficult and costly to implement, reducing the advantages gained by hardware reuse and commodity microprocessors. Likewise, hardware-implemented techniques impact performance by forcing coverage on all applications even though it may not be needed. Thus, software-implemented techniques are now being looked at as potential solutions to the problems of unreliable hardware.

Software-implemented fault-tolerance techniques are attractive because they are inexpensive, portable, and flexible. It is much more cost-effective to implement fault-tolerant algorithms in software than hardware. Most vendors reduce costs and development time by reusing existing proprietary and third-party components. Incorporating fault-tolerance would complicate reuse and increase the necessity for the development of unique logic circuits. Software, however, runs on existing commodity components, and if well-designed, can easily be modified to accomodate post-production changes in a hardware platform. It can also be ported to other platforms for fault-tolerant heterogeneous computing, an important aspect in the current cluster-based supercomputing environment. Unlike hardware-implemented solutions, it also gives users the ability to forego fault-tolerance when it may not be necessary, such as during the frequent edit-compile-test cycles of application development.

However, software-implemented fault-tolerant techniques have been frequently criticized for their lack of transparency, implementation difficulties, and efficiency. Presently, the application programmer must bear the burden of implementing software-based fault-tolerant techniques. Not only does this increase the already difficult job of the parallel programmer, but it also reduces potential gains in reliability due to design and implementation faults. Software techniques also steal precious CPU cycles, reducing performance by a potentially considerable margin. In spite of these problems, examples exist that prove such techniques can be very useful [7, 11]. In fact, software-implemented fault tolerant techniques are now being adopted by many MPP vendors. For example, simple checkpointing was incorporated by Meiko into its CS-2HA system, and IBM incorporated a well-known checkpointing package, called Condor [11], into its SP2's LoadLeveler [9]. However, there is presently no complete software-implemented fault detection, reconfiguration, and recovery package for MPPs.

This paper presents a novel software-implemented fault-tolerance architecture, called the **Software-Implemented, Parallel-System Fault Tolerant** (SPF) layer, which is being built for the purpose of enhancing MPP reliability. This paper is organized as follows. The following section presents terminology and assumptions. Previous work is presented in Section 3. An overview of the architecture is presented in Section 4, followed by a description of the implementation of a prototype layer in Section 5. Section 6 presents preliminary results, followed by the conclusion in Section 7.

2 Terminology and Assumptions

2.1 Relevant Fault Assumption

The fundamental purpose of the SPF layer is to protect user applications, running on MPPs or workstation clusters, from hardware failures. Hardware failures result from transient, intermittent, or permanent hardware faults that manifest themselves as errors in software or firmware programs which cause erroneous and unexpected behavior [15].

The SPF layer has been designed under the single fault assumption. It was decided that the probability of multiple simultaneous faults is sufficiently small that protection against them would not justify the added complexity and overhead that would ensue. However, this does not imply that the currently designed SPF layer will not handle multiple faults. Certainly, if the faults occurred in different parts of the system, within different fault containment regions, it is probable that the layer will handle them both. Regardless, the SPF layer should still impart a significant improvement in reliability over systems without fault-tolerance, even under the single fault assumption.

2.2 Parallel Systems and Programs

Throughout this paper, the terms *parallel systems*, *multicomputers*, and *massively parallel processors*, all refer to the class of MIMD, distributed memory, message-passing, parallel computing systems. It should also be recognized that a cluster of workstations may also be viewed as a single multicomputer. In fact, the term *SCANs* (Supercomputers at Night) [4] has been coined to describe clusters of workstations that are programmed for and utilized as a single multicomputer. SCANs are becoming more popular through the proliferation of software libraries, or software platforms, that provide programming interfaces and facilities similar to those found on multicomputers.

The terms *application* and *program* are used to refer to a user's code, of either sequential or parallel variety, that is complete and self-contained. On the other hand, *task* is used to explicitly designate a program that is currently executing on a single node, and may be either a stand-alone sequential program, or one part of a parallel program.

3 Other Research

Researchers at AT&T have developed a set of software tools for the detection of and recovery from faults within sequential UNIX applications that cause them to crash or hang [7]. The tools `watchd`, `libft`, and `librepl` protect user applications by maintaining non-executing, shadow processes that can take over when the primary fails. `watchd` is a watchdog daemon that detects application failures through periodic "I'm alive" messages. If an application fails to respond within a certain timeout period, it is assumed to have failed and it is either restarted from a previous checkpoint, or its backup is initiated. Checkpointing is initiated through user inserted function calls, found in `libft`. In addition, other functions exist which allow the designation of volatile data to be protected, similar to the well known Recovery Box technique. In addition, automated file replication is provided through the efforts of routines found in `librepl`. Overall, these techniques have been proven to be very cost-effective, useful, and efficient. However, the tools make no attempt to handle the detection and recovery of failures which do not cause a crash or hang.

Researchers at the University of Wisconsin - Madison, have developed a software package for the transparent checkpointing and migration of sequential programs running on UNIX, called *Condor* [11]. Condor was primarily written to support long-running workstation batch jobs, where it is often necessary to migrate, pause, or restart jobs as network load demands. Condor checkpoints and migrates processes entirely outside of the UNIX kernel, by providing its own standard C libraries which control the checkpointing and migration of the host application. When a checkpoint

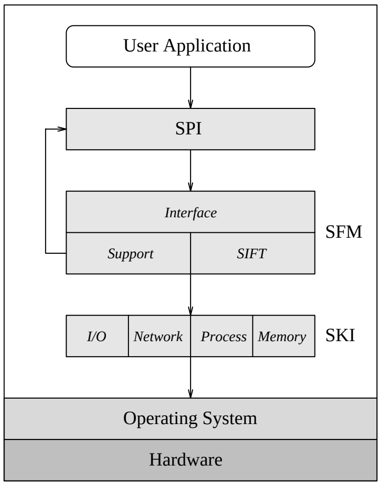


Figure 1: Block Diagram of the SPF Layer

is to be taken, Condor issues a `SIGTSTP` to the host application which is caught by a signal handler that saves the volatile state of the process. The handler then issues a `SIGQUIT` which causes a core-dump. When the application is restarted, initialization code reconstructs the state of the process using the checkpointed volatile state and the application's core-file If the application has been migrated to a different workstation, its environment may have changed such that it no longer has access to resources that it needs. To avoid this, the C libraries perform RPC calls to a concurrently executing shadow process on the original workstation, so, it appears to the migrated application as if it is still executing on the original workstation. Condor's concepts are, in many ways, analogous to those of the SPF layer. However, Condor's capability is limited to checkpointing and restarting of sequential applications.

There is currently an effort, based in Europe, to develop fault-tolerant MPPs using combined hardware and software approaches. The project is called *Fault Tolerance in a Massively Parallel System (FTMPS)*, and is supported by the ESPRIT-project 6731 [3]. The group's research, so far, has been focused on identifying and defining the overall required aspects of a successful fault-tolerant MPP. However, only preliminary surveys of fault-tolerant techniques have yet been released by the group. The target platform for their work is a Parsytec GC, but it is unclear what hardware modifications will be made to enhance its fault-tolerant capabilities.

4 The SPF Layer

4.1 Design Overview

The SPF layer is composed of multiple software sublayers that handle the various tasks necessary for fault tolerance. A block diagram depicting the SPF layer on a single processing node is shown in Figure 1. Although the diagram only shows one processing node, the layer does not reside on a single processor as a central controller, but is symmetrically distributed onto each node of the parallel processor. Parts of the SPF layer envelope the user application, joining with it at compile time, so that the layer may acquire and maintain control of its execution and unlimited access to the its code and data space. Thus, at execution time, the SPF layer is transparently distributed onto each node as part of the user's application. Other parts of the layer take the form of separate daemon processes that are either transparently created and destroyed with the application, or are statically loaded onto each processing node at boot time.

There are numerous advantages to the symmetrically distributed, multi-tiered design of the SPF layer. Distributed control is preferred over centralized control because it removes a potential single point of failure. In addition, the probability of fault propagation is reduced because each node is a self-contained fault isolation area. Better performance is also achieved because a central controller is a potential performance bottleneck. The multi-tiered approach enhances portability by separating machine dependent code from machine independent code. This code separation also makes it possible to replace or upgrade a software module without recompiling the entire layer, and has the potential for implementation as dynamically loadable run-time modules. The multi-tiered approach also facilitates concurrent development of the layer itself, by grouping functionally similar code into modules with well defined interfaces. This modularity also reduces the probability of software design and implementation faults by providing isolation areas for the detection and correction of errors during debugging.

4.2 User Application

The user application resides on top of the SPF layer, interfacing with the SPI sublayer. It should be noted that in the diagram "user application" refers to either a solitary sequential program or one task of a larger parallel program. A user application is a composite of user code and library code provided by the system for managing parallel programming specific, or system specific, tasks. For instance, the nCUBE2 series of systems provide libraries of routines that comprise the parallel programming interface and facilitate such things as message passing (`nread()`, `nwrite()`), resource allocation (`nopen()`, `nclose()`), and the loading of programs (`rexecl()`). A user application that is written for use on the nCUBE2 is not portable to another system if it calls the nCUBE2 specific routines. These calls must be intercepted by the SPF layer in order to ensure consistency, which is managed by the SPF Program Interface.

4.3 SPF Program Interface (SPI)

The SPF Program Interface sublayer has two main responsibilities: to preprocess the application source code and to act as a translator between the application and the SPF layer. The SPI consists of two parts, a definition file and a code library.

The *definition file* is used to preprocess the application source code and replace selected standard library function calls with calls to routines in the SPI code library, or insert calls at predefined places that will bounce control into the SPI code library. The calls that the preprocessing phase replaces will typically be those necessary for abstracting the underlying architecture, such as a routine that returns the current node number the process is running on. In addition, calls to routines that are to be handled by the SFM sublayer for the purposes of fault-tolerance, such as standard I/O routines, may also be replaced. Calls that may be inserted include calls into assertion testing routines, or triggers that bounce control into the SPI code layer for purposes such as control flow checking and signature generation. These calls may be inserted by the user as well.

The facilities that actually perform the code conversion are not explicitly specified as part of SPI. For most systems and languages, a standard preprocessor is provided which processes user code prior to compilation, performing macro substitutions and extricating unnecessary code according to embedded directives and separate preprocessing definition files. For these systems, the SPI definition file can be implemented for use with the standard preprocessing facilities provided. However, if such facilities are not provided, then they could be provided along with the SPF layer as part of the implementation for the particular system in question.

The *code library* contains routines that map function calls, at run-time, between the user application and the SFM module. The application is linked against these routines at compile-time, and execute as part of the application. A major portion of the code library will be routines that map system specific calls to more general calls into the SFM module.

For example, an nCUBE2 program calls `whoami()` to determine the distinguishing ID of its node among the set of nodes in its hypercube, as well as other information like the process ID of a controlling host routine, the dimension of the current hypercube, and the calling process's own ID. Since `whoami()` is an nCUBE2 specific function call, it would not be wise for the SFM module to incorporate it as part of its interface, since some of the information it provides, like the dimension of the current hypercube, is not pertinent for other systems. However, establishing a node's identification is generally provided on most other systems. Thus, the SFM module might provide a function `node_id()` that returns only the ID of the current node. The SPI code library would then be responsible for mapping calls for `whoami()`, to the SFM module's `node_id()`. Thus, both portability and efficiency are gained.

Also included in the library will be routines to handle programming language library calls, and calls to SPF specific routines, such as those that initiate checkpoints, perform assertion testing, and monitor control flow. Calls to SPF specific routines may either be inserted by the user, or automatically by the preprocessor, or not at all. These routines are provided more as a convenience than a necessity, since the layer is designed to provide fault-tolerance transparently. However, efficiency is often improved when the application programmer decides when checkpoints should be made, or assertions tested. Thus, the SPI code library provides an interface into the SFM module for use by the application programmer.

4.4 SPF Fault Tolerance Module (SFM)

Design

The SFM module is the primary driver of the SPF layer. It's primary responsibility is to envelope and control a user's application as required by the SIFT algorithms it implements. The decision of which particular algorithms it implements are left up to the programmer of the SFM module and not explicitly defined by the SPF layer itself. The layer was designed this way because no one fault-tolerant technique is applicable to every application and every computing system. For instance, the fault-masking capability of a costly TMR scheme is not necessary for high-availability applications where an occasional failure is acceptable, as long as reconfiguration and recovery is quick. Likewise, an application may not always need protection. Thus, the SPF layer is designed to provide a framework in which modules may be crafted to meet the needs of all the different types of applications and computing systems. However, rules and requirements regarding the design of the modules must be defined and followed in order to ensure compatibility with the rest of the layer. Thus, the basic design of the modules is defined as part of the SPF layer. An SFM module is divided into three submodules: the interface submodule, the SIFT submodule, and the support submodule.

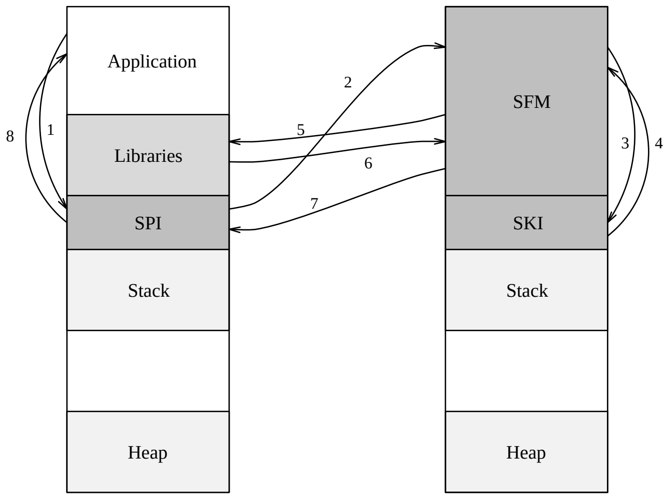
The *interface submodule* contains routines which handle function calls from the SPI sub-layer. The calls coming from the SPI layer can either be redirected system or language library calls, or user/compiler inserted SPF function calls. Calls that are not critical to the SIFT algorithms implemented by the SFM module may only be wrappers that pass control to the intended function without performing any extraneous operations. Although this exacts a small performance penalty, it increases flexibility by allowing modules to be dynamically loaded at run-time, instead of statically linked at compile time.

The *SIFT submodule* contains the routines that implement the SIFT algorithms for fault detection, reconfiguration, and recovery. No assumptions are made as to what algorithms are implemented, but all implementations should use the facilities provided by the SPF layer as intended in order to ensure stability and security of the layer itself.

The *support submodule* contains a core of routines that should be consistent across all modules. These routines are responsible for miscellaneous startup and shutdown chores; daemon creation and destruction; construction and management of the SFM module's stack; construction and management of the user application's as well as the SFM module's heap; and other as yet undefined tasks such as the dynamic loading of submodules and the manipulation of user threads.

Operation

At run-time, the SFM module and the user application execute as coroutines, as shown in Figure 2. At startup, SPI acquires control from the operating system and immediately passes it on to the SFM module before the user application has a chance to execute. During the course of the SFM module's assorted initialization chores, control is passed to and from the SPF Kernel.



!. During execution, the application places a function call, which has been redirected into the SPI
2. After swapping stack pointers, the SPI calls the the SFM for the intended function call
3. The SFM may then call the SKI
4. The SKI returns control to the SFM
5. After necessary SIFT actions, the SFM calls the library function
6. The library then returns control to the SFM
7. After more SIFT actions, the SFM returns control to the SPI
8. The SPI then returns control to the user application

Figure 2: Flow of Control in and Application Using SPF

After initialization is complete, the SFM module passes control to the user application by calling its startup routine. As the user application executes, it periodically calls into the SFM module via the SPI sublayer. The SFM module will then perform actions demanded by the SIFT algorithms implemented for the call, which often includes placing the function call for the user application. Since the system and language libraries are linked in as part of the user application, when the SFM module is placing the function call it is in essence calling back into the user application. Thus, the two can be thought of as coroutines.

The SFM module maintains its own stack and heap, shown in Figure 2, as well as the user application's heap, in order to reduce tainting, prevent resource usage problems, and detect leaks and overruns. For architectures that support it, the SFM module may be implemented as a thread.

The support submodule contains routines which construct a separate stack within the heap memory segment for the SFM module. A separate stack is convenient and possibly necessary for SIFT algorithms that investigate or alter the user application's stack. For example, some fault-detection routines generate signatures over the stack frames in order to detect errors in control flow or data, but SFM frames present on the stack could taint a signature and cause a comparison to fail. Likewise, some rollback schemes restore the stack to a previous state and reexecute when a failure is detected, which could cause SFM frames to be destroyed. Another reason for maintaining a separate stack is to avoid problems with resource limitations. For example, most operating systems impose a limit on stack size that might be exceeded by the SFM layer if it executes on top of the user application's stack. However, heap limits tend to be larger and more controllable. Thus, moving the SFM module's stack into the heap reduces the possibility for failure due to system imposed limitations.

Switching between the application and SFM stack is performed by the SPI layer. When the SPI layer intercepts a call from the user application, it immediately saves and replaces the contents of the stack pointer register with the SFM module's stack pointer. When it calls into the SFM interface submodule, its frame is forced onto the SFM module's stack. The top of the user application stack, however, contains only the parameters to the SPI call and any automatic variables it requested. When the SFM module returns control to the SPI layer, the layer swaps the stack pointers again before it returns control to the user application.

The support submodule routines also manage heaps for both the user application and the SFM module. Maintaining separate heaps allows the SFM module to keep track of which blocks within the heap segment is allocated to the user application. This is important information for many of the SIFT schemes, such as checkpointing. It eliminates the chances of stack tainting, and reduces the probability of problems due to resource limitations. It also provides the means to detect leaks and corruption, common software faults that cause very non-deterministic errors which could be wrongly assumed to be caused by hardware faults. Upon startup, an initialization routine is called which creates free lists for both heaps, dividing the default heap segment between the two lists and the SFM stack. When the user application issues a call requesting heap space, the SPI intercepts it and directs it into the SFM module which allocates blocks from the user application's heap free list. When the SFM module needs heap space, it allocates blocks from its own free list. When the default heap segment is full, the SFM module requests a larger segment from the system. Conversely, it returns memory to the system by reducing the segment when it is larger than necessary.

4.5 SPF Kernel

The SPF Kernel is a collection of modules that contain routines for handling low-level manipulation of communications, I/O, processes, and memory. Each module is a set of small, frequently used, basic routines that will typically already be provided in the operating system. The separate components of the kernel are the metwork management module, I/O management module, process management module, and memory management module.

The *network management* module is responsible for handling interprocessor communication, as well as maintaining a database network state information. In order to communicate in the presence of faulty nodes or links, the network management module takes control of the message routing if a single failure has been detected. Numerous algorithms for routing messages in the presence of faults have been developed for various network topologies, and the particular method used depends on the topology for which the SPF layer resides. The maintenance of a database of information regarding the state of nodes and links in the system is imperative for routing messages and reallocation of processes. Although primarily used as a guide for message routing, it also serves as the basis for such tasks as the selection of a spare node for reallocation and recovery of a failed process. Since the SPF layer is architecturally independent, the network management module must provide this service.

The *I/O management* module is responsible for maintaining the consistency of files and other input/output streams. One of the primarily tasks of the module is to ensure that redundant tasks do not try to modify the same stream. Such action would result in unpredictable behavior. Thus, the module's primary task is to intercept writes issued by redundant processes. For example, to perform a write to a file that is being used by redundant processes peer I/O modules, first synchronize the redundant processes. The I/O module from the primary process may then issue the write on the file. The return value is then broadcasted to each process. This assures that there is only one write issued onto the file and that each process gets the correct return value.

The *process management* module is responsible for controlling or accessing local or remote processes. Since process manipulation typically requires protected mode access, special operating system routines are used that are typically reserved for debuggers. For example, the UNIX `ptrace` command allows one process to perform operations such as pausing, restarting, and killing another process. It also provides a means of accessing and modifying register and memory contents. These capabilities are imperative to many fault tolerance techniques that perform direct memory or register operations such as signature generation and checkpointing.

The *memory management* module is responsible for explicit manipulation of memory contents, including the mapping of addresses in a distributed shared memory configuration. However, this is only possible if the handling of the address mapping is not handled implicitly by the operating system.

5 Implementation

In order to investigate the technical feasibility, efficiency, and coverage of the SPF layer, a subset of the layer was implemented for testing. The SFM module used in the minimal layer implemented a simple parallel, duplexed algorithm [15]. The layer was developed for use with MPI running on a pair of networked Sun Sparc 1 workstations running SunSoft's SYS V compliant Solaris 2.2 operating system. The minimal SPF layer was written in strict ANSI C, conforming to POSIX.1 conventions and following GNU guidelines for portable programming, comprising 9,126 lines of code spread over 281 files.

5.1 MPI Overview

MPI (Message Passing Interface), was drafted by the MPI Forum, a committee of over 50 researchers from over 40 organizations, which began meeting in November, 1992 [4]. Among the Forum's members, are the researchers behind PVM, PARMACS, P4, PICL, nCX (nCUBE's microkernel), and Express. The primary objective was to establish a standard which would focus the

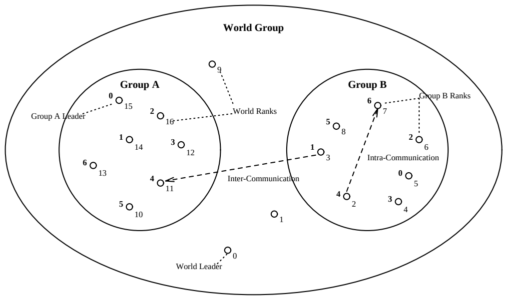


Figure 3: Schematic of Process Groups in MPI

message-passing platforms, currently using a myriad of proprietary and incompatible libraries and interfaces, down to one well-designed and well-defined interface.

MPI uses the concept of groups to represent parallel programs in its message passing paradigm, as shown in Figure 3. Processes are lumped into *groups* within which they can communicate and collaborate. *Ranks* are used to distinguish between processes within a group for the purposes of communication or application specific ordering. Processes within a group communicate with each other by addressing messages with ranks. Before communication can proceed, each process must acquire a communicator. A *communicator* maintains contextual information about the communication within a group. More information about MPI can be found in [4].

5.2 Implementation Detail

5.2.1 SPI

The SPI sublayer is responsible for preprocessing user applications for the ultimate purpose of redirecting calls through the SPI code module and into the SFM module.

This implementation relies on the preprocessing services provided by the standard C compiler's preprocessor. The subset of function calls to be intercepted are replaced by calls into the SPI code library through C macro substitution. These macros are all contained in a single header file, `spf.h`. Thus, the only effort required to incorporate the SPF layer into an application is to include in each source file spf.h, compile, and link it against the SPF libraries.

The code library contains routines for each of the intercepted function calls. Each routine in turn calls `sfm_fcall()`, the interface provided by the SFM interface submodule for redirected function calls. This call is wrapped by assertions to detect an erroneous return value, as well as routines which save, correct, and restore global volatile state variables which the application program may depend upon, such as errno.

5.2.2 SFM

The SFM sublayer is responsible for the implementation of the duplexed algorithm, and is divided into the interface, SIFT, and support submodules.

Interface Submodule

The interface submodule exports the following calls:

```
spf_return_t sfm_fcall(void *fcall_ret, spi_fcall_id_t fcall,...)
spf_return_t sfm_trggr(void *trggr_ret, spi_trggr_id_t trggr,...)

void sfm_assert(int result, char *file, int line)

spf_return_t sfm_wrap_state_save(void)
spf_return_t sfm_poc_state_restore(void)
spf_return_t sfm_poc_state_save(void)
spf_return_t sfm_wrap_state_restore(void)
```

`sfm_fcall()` is the entry point for all redirected calls from the SPI code library. It's parameters `fcall_ret` and `fcall` hold a pointer to the return value of the function call and a unique identifier which is used to select the address of the actual function call, respectively. The variable argument list contains the function call's arguments. A single entry point was used to simplify the detection of faults propagating across the boundary of the SFM fault-containment region. It also simplifies formal specification, and provides greater flexibility.

`sfm_trggr()` and `sfm_assert()`, are provided in this implementation as example user-inserted/compiler-inserted interfaces into the SFM module routines. `sfm_trggr()` is implemented identically to `sfm_fcall()`, allowing the SFM module programmer the ability to provide a variable number of facilities to the application programmer, without modifying the module's interface. This is analogous to the versatile `ioctl()` interface provided within UNIX. Although `ioctl()` is met with varying degrees of emotion in the UNIX community, it has served very well to expand the number and types of devices supported by the UNIX kernel without requisite modification to its interface. `sfm_assert()` is a simple boolean assertion tester. Arguments for it's `file` and `line` parameters are intended to be provided by the `__FILE__` and `__LINE__` C macros.

The four state preserving functions, `sfm_wrap_state_save()`, `sfm_poc_state_restore()`, `sfm_poc_state_save()`, and `sfm_wrap_state_restore()` are used to preserve global volatile state. `sfm_wrap_state_save()` is used at the point of wrapping by the SPI code module. The saved state is then restored at the point-of-call, in the SFM module, by `sfm_poc_state_restore()`. This is necessary if the function call depends on the value of the global state. After the call is placed, the state is saved by `sfm_poc_state_save()`, to be restored at the point of wrapping by `sfm_wrap_state_restore()`. For example, `errno` is set by system function calls. If the user application checks `errno` after a system call, and that call has been redirected, then its contents may have been corrupted by system calls in the SPF layer. Thus, if `errno` is not preserved at the point-of-call and restored just prior to return, the application program will be in an erroneous state induced by the SPF layer.

Support Submodule

The routines in the support submodule are primarily for initialization and shutdown of the SPF module. At startup, initialization routines within the support submodule are responsible for initiating and establishing the MPI communication groups that are used to define and separate the duplexed parallel applications. To create duplexed parallel applications, a group twice as large as necessary is created at startup. After `MPI_Init()` has been called, the support submodule's initialization routine, `split_comm()`, splits the *world group* in two, creating separate groups with separate communicators and identical numbers of processes in each, like *Group A* and *Group B* shown in Figure **??**. The communicators for each group is then subbed for the world's `MPI_COMM_WORLD` communicator in calls placed by the application. The SPF layer uses the `MPI_COMM_WORLD` when communicating between the groups.

SIFT Submodule

The sift submodule contains routines that implement the duplexed, distributed voting, fail-fast fault detection algorithm. The collection of routines can be separated into two general groups: I/O control and voting.

When parallel applications are duplexed, it is imperative that only one application's I/O is posted. The SIFT submodule ensures consistent I/O by intercepting and snubbing the output of one of the duplexed applications. Thus, in this implementation, routines like `write()` and `fputs()` are handled by the SFM module, such that only one call is placed between the two, but the return value is made available to them both. Likewise, prior to the posting of any output, the data is voted on by both of the duplexed applications.

Voting between the duplexed applications is initiated on any data input or output, including messages. When an application posts a message to be sent, the SFM module generates a 32bit CRC over the entire contents of the message, including the destination address. The CRC implemented in this layer is the same as that used as the frame check sequence in ADCCP (ANSI X3.66, also known as FIPS PUB 71 and FED-STD-1003, the U.S. versions of CCITT's X.25 link-level protocol). Each application than sends its CRC to its peer in the duplexed application, and they both compare CRCs simultaneously. If either application detects a difference, then the application is said to be in error and immediately terminates. Thus, it is "fail-fast."

Program	Mean Execution Time Unprotected	Mean Execution Time Protected	Overhead
FFT	386.74	454.61	17.6%
nQueens	19.23	25.48	32.5%
Pi	35.47	41.22	16.2%

Table 1: Performance Comparison of Test Applications Using/Not Using SPF

5.2.3 SKI

Implementation of the SPF Kernel for the minimal SPF layer was not necessary, as was hoped for in the design of the SPF layer. All of the routines necessary for implementing the duplexed SIFT scheme were either provided by the operating system and MPI, or implementable within the SFM and SPI sublayers, without resorting to system specific calls or manipulations.

6 Results

6.1 Performance

In order to estimate performance, the minimal SPF layer was tested using a set of sample programs. Since performance of the layer is highly dependent upon the particular SIFT scheme used in the SFM module, the duplexed SIFT scheme implemented was chosen not only for its quick fault detection capabilities, but also for its potentially high overhead. Thus, the performance measurements obtained should represent an approximate upper-bound, with more conservative SIFT schemes incurring considerably lower overhead. Certainly, few realistic applications will need the fault detection that a duplexed scheme provides.

Most of the measurements were made on a pair of dedicated Sun Sparc-1s running Solaris-2.2, each with 16MB of memory and 32MB of local swap, interconnected by a 10 Mb/s BNC Ethernet network.

The sample programs chosen for testing were:

- *FFT* computes a FFT over a file of 65,536 complex floating point numbers.
- *nQueens* computes the 92 solutions of the infamous 8x8 nQueens problem.
- *Pi* computes a Monte Carlo approximation to using 1,000,000 iterations.

Table 1 shows the results of 1,000 executions of each of the sample programs with and without the protection of the SPF layer. These results were acquired on 8 Sparc-1, Ethernet networked workstations. The experiments not using the SPF layer were run on 4 workstations,

A	B	C	D	E	F	G	H	I
Method	Rate	Area	Exps	No Effect	Masked	OS Detected	SPF Detected	Undetected
Single	NA	Heap	100	65	14	0	21	0
Single	NA	Stack	100	71	23	0	6	0
Single	NA	Text	100	88	0	12	0	0
Random	0.1	Heap	100	0	0	7	93	0
Random	0.1	Stack	100	63	0	30	7	0
Random	0.1	Text	100	0	0	100	0	0

Table 2: Results of Fault-Injection Experiments for a Parallel FFT using SPF

and those using the SPF layer were run on all 8 workstations. The FFT was modified to compute 100 transforms of 65,536 bytes each in order to stress the SPF layer and increase execution time for Although the overheads are not within generally acceptable bounds, which is typically 10% - 15%, they are good considering the nature of the duplex scheme.

Overall, the performance results were very encouraging. As was stated previously, large overheads were expected because of the nature of the duplex SIFT scheme implemented. However, the results were much lower than anticipated, especially considering the high message passing penalty of the MPI workstation platform.

6.2 Coverage

In order to estimate coverage, fault-injection experiments were performed using a fault-injection tool being developed for this research, called Flip-It. Flip-It uses the /proc file system to access the image of a running process, inserting faults as defined by the user. The /proc file system is currently available on only a few UNIX systems, like Solaris and Linux, but will most likely become a standard part of the SYS V directory tree. Currently, Flip-It can only insert random transient bit faults, and pseudo intermittent bit faults, but work continues to incorporate much more functionality.

The results of the fault-injection experiments are shown in Table 2. These results were obtained with a two-task parallel FFT program using the SPF layer on four workstations. In all of the experiments, only uniformly distributed, simulated transient bit faults were injected. Two sets of experiments were performed and are labeled *Single* and *Random* in the table. For the Single experiments, a single fault was injected at the beginning of the program's execution and monitored until an event of interest occurred or the program completed successfully. For the Random experiments, faults were injected periodically, at a rate of one every ten seconds, until an event of interest occurred. For the Random experiments, faults were not monitored or tracked.

Events of interest are shown in columns G and H. Column G contains the count of faults which the operating system detected before the SPF layer. Faults caught by the operating system should not be considered a sign of deficiency in the SPF layer. The purpose of the SPF layer, in the duplexed configuration at least, is to detect faults which would otherwise go unnoticed and cause erroneous output. Column H contains those faults which the SPF layer detected, either through voting or internal assertion testing. If the fault was overwritten, it was said to have been Masked. Column F contains the count of faults that were masked. It should be noted that although a fault was masked, it could have still caused an error which may remain latent until later in the program, where it surfaces as a failure. Thus, a masked fault is not an event of interest, because the experiment should not be terminated when it occurs. If the fault remained latent throughout the duration of the program then it was said to have No Effect, and the experiment was marked off as insignificant. Column E contains a count of the insignificant experiments. However, if the error surfaced later and the SPF layer failed to detect it, then it was an Undetected Error. Column I contains a count of the number of undetected errors. Undetected errors were detectable post-mortem by comparing a program's output to a correct copy.

Clearly from the table, the SPF layer performed very well in detecting faults. Throughout the 600 experiments, no errors escaped undetected. As was expected, faults in the text region were most often caught by the operating system, with faults in the heap and stack areas being caught more often by the SPF layer. Although it is obvious that more experiments must be performed in order to establish a statistically significant measure of the layer's coverage, the preliminary results are very encouraging.

7 Conclusion

A novel architecture for a software-implemented fault-tolerant layer for application reliability on massively parallel computers and distributed computing systems was presented. The performance results were encouraging, considering the nature of the duplex SIFT scheme implemented. The layer performed well in detecting faults, and demonstrated that user-level software-implemented can be resonably efficient and effective.

Bibliography

[1] Convex Computer Corp., "Convex Exemplar SPP1000 Technical Summary", Report #080-002322-000, 1994.

[2] Cray Research Inc., "MPP Technology Preview", 1992.

[3] G. Deconinck et al., "Fault Tolerance in Massively Parallel System," ESPRIT-Project #6731 Report, 1994.

[4] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, MIT Press, Cambridge, Massachusetts, 1994.

[5] G. Holland and D. Pradhan, "Fault Tolerant Multiprocessor Systems," Report #94-05, Reliable Computer Tech., Sept. 1994.

[6] R. Horst, "Massively Parallel Systems You Can Trust," *COMPCON94 Digest of Papers*, San Francisco, Feb. 1994.

[7] Y. Huang and C. Kintala, "Software Implemented Fault Tolerance: Technologies and Experience," *Proc. 23rd Int. Symp. on Fault-Tolerant Computing*, Toulouse, France, June 1993.

[8] IBM Corp., "Scalable POWERparallel Systems: High Performance Technical Computing Solutions," Report #GH23-2485-01, 1994.

[9] IBM Corp., "LoadLeveler 1.2 for IBM Scalable POWERparallel Systems," Software Announcement, August, 1994.

[10] Intel Corp., "Intel Paragon Supercomputers," Report #203/004, 1994.

[11] M. Litzkow and M. Solomon, "Supporting Checkpointing and Process Migration Outside of the UNIX Kernel," *Proceedings of the Usenix Winter Conference*, San Francisco, California, 1992.

[12] R. Harper and J. Lala, "Fault-Tolerant Parallel Processor," *Journal of Guidance, Control, and Dynamics*, vol. 14, no. 3, pp. 554-563, AIAA, May-June 1991.

[13] Meiko, "CS-2HA High Availability: Technical Description," Report #X0126-00B 100.01, 1994.

[14] nCUBE, "nCUBE 3: Technical Overview", Foster City, CA, 1993.

[15] D. Pradhan ed., *Fault-Tolerant Computing*, Prentice Hall, 1995.

[16] Sequoia Systems Incorporated, *Series 400 Technical Summary*, June, 1992.

[17] H. Simon, "High Performance Computing: Architecture, Software, Algorithms." Report #RNR-93-018, NASA Ames Research Center, Dec. 1993.

[18] G. Zorpette, "The Power of Parallelism," *IEEE Spectrum*, Sept. 1992, pp. 28-33.

[19] Stratus Computer Incorporated, "XA/R Series 300," Stratus Product Report, 1992.