# Application-Level Fault Tolerance in the Orbital Thermal Imaging Spectrometer

E. Ciocca, I. Koren, Z. Koren, C.M. Krishna
University of Massachusetts, Amherst, MA

D. S. Katz
Jet Propulsion Laboratory, Pasadena, CA

## Abstract

*Systems that operate in extremely volatile environments, such as orbiting satellites, must be designed with a strong emphasis on fault tolerance. Rather than rely solely on the system hardware, it may be beneficial to entrust some of the fault handling to software at the application level, which can utilize semantic information and software communication channels to achieve fault tolerance with considerably less power and performance overhead. This paper details the implementation and evaluation of such a software-level approach, Application-Level Fault Tolerance and Detection (ALFTD) into the Orbital Thermal Imaging Spectrometer (OTIS).*

## 1. Introduction

This paper considers an application that is intended to run in a hostile environment - the Orbital Thermal Imaging Spectrometer (OTIS), part of NASA's Remote Exploration and Experimentation (REE) [1] project which focused on the implementation of a distributed system in orbiting, space bound, or even extra-terrestrial environments. OTIS does not have any inherent methods of fault detection, nor methods for tolerance of component-disabling faults (beyond its dynamic allocation scheme). Fault detection and tolerance would need to be implemented either in the underlying hardware, or in lower-level software such as the OS. While these may be able to compensate for obvious faults, such as a failed processor, they may not help when subtle errors like data corruption arise.

By entrusting an amount of fault detection and tolerance to the application itself, redundancy can be more efficiently utilized. By knowing the type of output which should be produced, deviant data can be suspected as faulty, and ad-hoc redundancy can be used to double-check the integrity of the suspect data and possibly provide the correct output, at the price of some runtime increases.

The paper briefly sketches the *Application Level Fault Tolerance* (ALFT) and *Application Level Fault Tolerance and Detection* (ALFTD) methodologies. It explores the types of data trends in OTIS that can be used for applying ALFTD by providing for fault detection and correction, and provides methods for fine-tuning the fault detection mechanisms. While this paper is limited to the exploration of a single application, the methodology used to apply ALFTD to OTIS provides insights which will be useful when applying ALFTD to other, similar, applications.

## 2. ALFTD vs. ALFT

A well-known software-based fault tolerance technique is Algorithm-Based Fault Tolerance (ABFT) whose primary function is to detect and correct errors at the word or data-structure level. ABFT was introduced in 1984 [2], with recent works on the topic which include [4] [3] [5]. For the most part, ABFT techniques have been geared toward providing fault tolerance in matrix operations in multiprocessor systems. To suit applications like OTIS with limited matrix operations, ABFT must be modified in order to be useful.

In previous research documented in [6], we described ALFT (Application-Level Fault Tolerance), a means by which to compensate for the failure of processing nodes in a real-time system without creating significant overhead or deadline misses. This is accomplished in software through partial redundancy, with neighboring nodes executing scaled-down versions of the primary processes.

We demonstrated ALFT's viability in several benchmark applications, by showing that software redundancy need not rely on mirror copies of processes. In a real-time target tracking application (RTHT) [7], faults disabling the "primary" process of a node could be almost completely tolerated by the run of a scaled down "secondary" process. These secondary processes could run with a calculation overhead as low as 17% of the primary's load and still produce viable results. Such a lightweight secondary incurs tolerably small runtime overhead, and can be run after the primary on a particular node has finished, thus eliminating the need for hardware redundancy.

The distribution of secondaries and primaries allowed for the tolerance of even processor-disabling faults. The typical distribution of primaries and secondaries is shown in Fig. 1. With a distribution such as this, if node 1 should become unusable, node 2's secondary process would still complete the work expected from node 1's primary process (at lower resolution). This model is used for applications that have distinct process workloads - work assigned statically to each node based on its identity. Further research with dynamic workload allocation has shown that such secondary identity is not always necessary, and that secondaries which dynamically distribute work can compensate for a fault with a more balanced overall runtime.

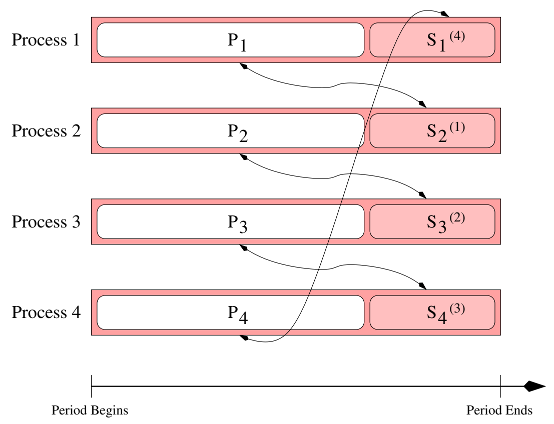


**Figure 1. A typical ALFT task allocation**

Previous ALFT work provided general methods by which to reduce the overhead of a secondary process.

- *Prioritizing*: The secondary is scaled by considering only the most important data. By ignoring unimportant data, only a part of the calculations need to be completed. This method assumes that the relative importance of data is inherent or easily determinable.
- *Functional Reduction*: The secondary is scaled by substituting the primary algorithm for faster, but less accurate, alternatives. This method would also preclude any other method of overhead savings through approximation of results (e.g., precomputed lookup tables).
- *Granularity Reduction*: The secondary is scaled by using only a predetermined fraction of the input data, such as every $n$th element in a matrix. The remaining data is compensated by interpolation between adjacent elements, or the result of one element is applied to a greater area of elements.

Similarly to ALFT, the ALFTD (Application-Level Fault Tolerance and Detection) methodology is based on the structure of primary and secondary processes seen in Fig. 1. However, the secondary process is invoked only if the primary process results in a suspect result. The secondary's recalculated result is then compared to the suspect result and a determination is made as to whether the output was caused by a transient fault or is a correct result which happens to be out-of-range.

ALFTD has two main advantages over ALFT. While an application with ALFT can survive total processor failures, the lack of sanity checks causes data corruption faults to remain largely undetected. Also, fault tolerance in ALFT is provided proactively, continuously invoking overhead, while ALFTD only requires secondaries to be run reactively when a fault has manifested itself. ALFTD's computation overhead is thus proportional to the fault rate, with some negligible constant overhead due to fault detection filtering. If faults are rare, ALFTD's calculation overhead will be consistently low. In cases of frequent faults, where the overhead of the secondary becomes more prominent, the calculation overhead is directly proportional to the granularity of the secondaries. Computation overhead is also dependent on the accuracy of the application's fault detection filters, as inaccurate filters can sometimes result in unnecessary task redundancy. This makes the calibration of filters important not only to an application's error minimization, but also to the reduction of its calculation overhead.

In order for ALFTD to detect potentially faulty outputs, one or more filtering "bands" are created. If the produced value is within some acceptance interval, it is considered valid, while if outside the interval, it is suspected to be invalid. The suspect data is then reprocessed at secondary resolution, and if the result is acceptable, it is assumed that the primary was faulty and the secondary result is used. If the secondary returns an approximately similar or a worse result, it is assumed that either the fault was a false alarm, or that the secondary has itself failed.

## 3. The OTIS Application

The Orbiting Thermal Imaging Spectrometer, a prototype application developed by the University of Washington for NASA's REE project [1], was selected for this study due to the high likelihood of environmentally induced errors during its execution. OTIS, as the name implies, is satellite-based software to be run in an orbiting distributed system. Using on-board sensors, radiation data is collected from the atmosphere, refined to provide temperature and emissivity data, and presented as a simple color-coded bitmap. The orbiting environment of this system makes it a target for potentially destructive radiation, so the necessity for accurate fault detection and tolerance is apparent.

OTIS utilizes the MPI library [8] for its network communication. MPI is open-source software, so it has been modified to accommodate a variety of platforms as well as underlying network hardware. The use of standard MPI calls allows code written for a general distributed system to be applied to the specific system of an orbiter, with no code alteration and approximately the same overall behavior. In OTIS, each physical processor may be allocated one or more tasks. Primary and secondary tasks are allocated distinctly.

Input to OTIS is in the form of a three-dimensional array. The $x$ and $y$ dimensions correspond to geographical space on the surface of a planet. The $z$ dimension is comprised of the same space measured in various wavelengths of radiation. OTIS's output comes in various forms. Immediately available are the numerical representations of temperature and emissivity data. Temperature, in Kelvin, is expressed as a two-dimensional array of floating-point numbers for the same $x$ and $y$ dimensions as the input. This data can also be expressed as a greyscale output in a bitmap. Emissivity data is in the same format and size as the input, and shows the radiation emitted by the scanned surface, minus ambient, atmospheric, and other forms of noise radiation.

OTIS' calculations are performed in a master/slave fashion. Each working node is assigned an ID number, with node 0 as the master. The master maintains a list of which rows (in the geographical $y$ dimension) are yet to be calculated, and allocates them one at a time to the slaves. Once a slave receives a row, it performs a series of calculations on the data. The calculated values are returned to the master, where they are stored in an output file, and also in a bitmap, if so configured. When it returns the results, the master, a reflexive process, assigns the next available row. This continues until the entire dataset has been computed, and the master sends a stop sentinel to all the slaves.

## 4. Applying ALFTD to OTIS

Even if we consider the master process to be fault-free, OTIS is still susceptible to faults in the slave processes. We consider two types of faults - the crash of an entire processing node, or an error caused to a single data pixel while it resides in memory, is being processed, or is being transferred over the network.

OTIS has a dynamic task allocation of its workload, and as such already has facilities for load rebalancing when an entire node crashes - the remaining nodes will do the extra work in the same first-come, first-served manner as in a non-faulty execution. Thus, the rest of this paper is devoted to detecting and correcting faults occurring in individual pixels.

We model the correct results of calculating one dataset as an array of $N_r \times N_c$ floating-point pixels $f(x_i, y_j)$. Each pixel has a probability $p_f$ of being faulty (independently of each other) and the error of a faulty pixel is a random variable.

ALFTD consists of two components - detecting an erroneous pixel, and reassigning the row in which the error occurred to a secondary process for (scaled down) recalculation. Although OTIS is not a hard real-time process and does not have strict deadlines, it must complete its work before the next batch of data arrives. Thus, for successful ALFTD implementation, these two components need to be performed quickly and with high efficiency.

### 4.1. Fault Detection

OTIS does not have an inherent way to detect errors in either the input data or output results, so no one acceptance test will have 100% accuracy. For good fault coverage, multiple tests applied in series on the output data are necessary. These acceptance tests (henceforth "filters") are application and data dependent. Each filter has two boundary values, the lower (left) bound and the upper (right) bound. Data outside these bounds is suspected to be faulty. Each boundary value can be set independently. In the case of OTIS, the results are expected to have two characteristics that will be present in non-faulty data - one in the data domain and one the gradient domain.

- *Natural Bounds* - The data represents a natural phenomenon (e.g., temperature, radiation), so it will mostly fall within an expected range. For example, while doing a temperature survey of the Earth, we should not expect to find anything too far below freezing, nor near the boiling point of water (with the exception of expected items like active volcanoes). When surveying a localized geographic area of only a few meters or kilometers, as a satellite would do, the cutoff values can be placed even tighter.

- *Spatial Locality* - The data within a small geographic area will change gradually. While dramatic sudden changes (hot or cold spots) are not unheard of, heat dispersion tends to average the spots into their surrounding area. Even so, the temperature within the spot is usually locally consistent, so the only inconsistent data will be along the spot's edges. This filter calculates the differences between neighboring pixels and expects them to fall within the acceptance interval (which will be symmetric around 0).

Clearly no fault filter of this type can be 100% accurate. Two types of mistakes can be made (resulting in two different penalties) - errors may be small enough to escape the filter (misses), or legitimate data may still be abnormal enough to trip even the most carefully calibrated filter (false alarms). A wide filter will result in more fault misses and erroneous data, while a narrow filter will result in too many false alarms which will cause the application to spend time recalculating already nonfaulty data, replacing it with secondary results that are inherently less accurate.

To find the filter bounds which strike the right balance between misses and false alarms, both faulty and non-faulty data pertaining to the application at hand must be investigated in order to gauge the probabilities of the two types of errors. In addition, the size of both penalties must be estimated to determine which is heavier and should be avoided as much as possible. If, for example, accepting an erroneous result as correct imposes a very high cost (much higher than recalculating a correct result), than we should select a filter (or a set of filters) which limits the probability of a miss to, say, ten percent, even at the expense of increasing the probability of a false alarm.

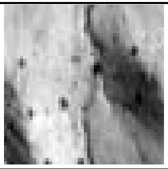
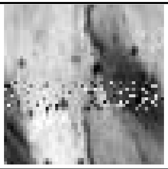
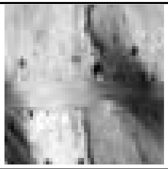
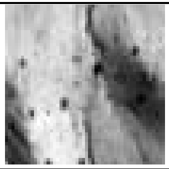
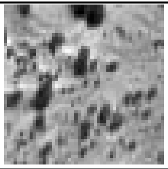
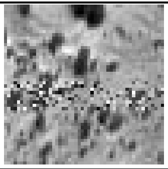
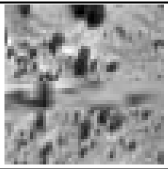
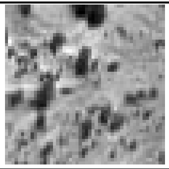
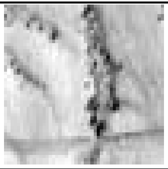
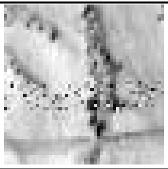
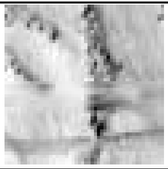
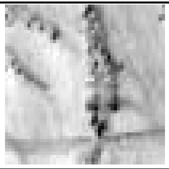
| | No Errors | Errors & No ALFTD | Errors & 25% ALFTD | Errors & 50% ALFTD |
|---|---|---|---|---|
| "Blob": This set was chosen for the broad areas of unchanging temperature, which is what we expect of natural data. |  |  |  |  |
| "Spots": Chosen for the obvious spots. It is more turbulent than stripe, and the turbulence spreads over a larger area. |  |  |  |  |
| "Stripe": Chosen for the swath of turbulent data through the center. Rapid changes in data foil the interpolation method. |  |  |  |  |

**Figure 2. Visual representation of three representative sets of OTIS data - errorless, with errors, and with error correction**

## 4.2. Secondary Scaling

Out of the possible methods of secondary scaling mentioned in Section II, the scaling most fitting for OTIS proved to be granularity reduction. Because the input is in the form of a matrix row, we can consider calculating only a few cells in the row and based on the spatial locality, calculate the missing cells by interpolating their calculated neighboring cells. This allows a reduction to 50%, 33%, 25% (and so forth) of a row's calculation overhead. The estimated temperature $F(x)$ of a skipped cell $x$ is obtained by linear interpolation between two calculated cells $x_1$ and $x_2$:
$F(x) = f(x_1) + \frac{f(x_2)-f(x_1)}{x_2-x_1}(x - x_1)$
To measure the error caused by a reduced granularity and subsequent interpolation for an array of $N_r \times N_c$, we denote by $f(x,y)$, $F(x,y)$, and $E(x,y)$ the correct reading, the value obtained through reduced granularity and interpolation, and the relative error at $(x,y)$. Thus
$E(x,y) = \frac{|f(x,y)-F(x,y)|}{f(x,y)}$
and the average relative absolute error over the whole array is $E_{avg} = \frac{\sum_{i=1}^{N_r}\sum_{j=1}^{N_c} E(x_i,y_j)}{N_r \times N_c}$

Clearly, the finer the granularity, the smaller the interpolation error at the cost of a higher calculation overhead. The next section presents numerical results pertaining to ALFTD in the OTIS application.

## 5. Numerical Experiments

Our numerical experiments focused on the two aspects of ALFTD in OTIS: ALFTD's ability to detect faults - i.e., distinguish between faulty and non-faulty data, and its ability to repair the faults quickly and with minimal error.

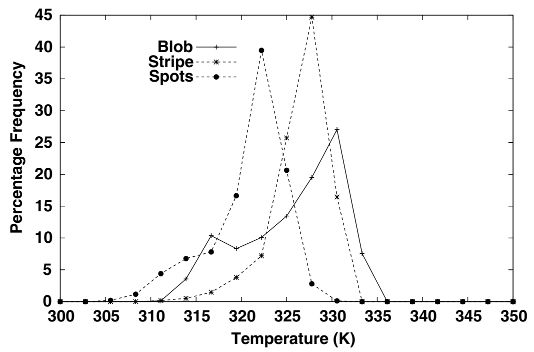


**Figure 3. Frequency of the data values for the three sample data sets (see Fig. 2)**

To calculate the probability of a false alarm for a given filter we must use statistics of correct data produced by the application, while for the probability of a miss we need to analyze faulty data.

As correct data, we selected three representative OTIS datasets (named Blob, Spots, and Stripe), shown as greyscale images in Fig. 2. For each of the three error-less datasets in Fig. 2 we calculated the frequency graph (Fig. 3) - used for the Natural Bounds filter, and the frequency of differences graph (Fig. 4) - used for the Spatial Locality filter.

The faulty data was obtained through simulation. If we simulated very large or very small errors, almost any filter will be successful. We, therefore, simulated the "worst case" for detection purposes - medium size errors. To generate the faulty data, we selected a set of $50 \times 50$ values out

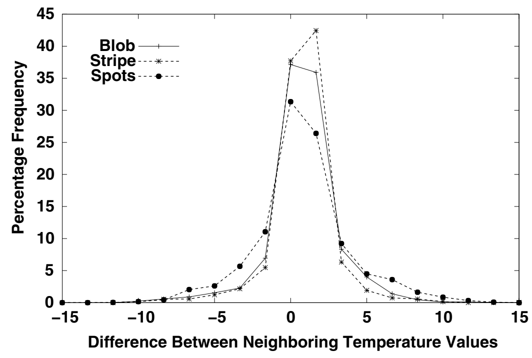


**Figure 4. Frequency of the differences between neighboring data values (see Fig. 2)**

of the "blob" set and injected errors with a probability of $p_f = 0.25$ into each pixel. The size of the errors was uniformly distributed between 1% and 5% of the correct value. The results discussed next describe averages over ten such experimental repetitions.

## 5.1. The Natural Bounds Filter

Fig. 5 demonstrates the trade-off between the misses and the false alarms probabilities, for various settings of the filter bounds. If, for example, a fault miss is very costly and the percentage of fault misses must be bounded by 10%, then the interval (314,321) will be selected at the cost of a very high false-alarm rate. If, on the other hand, false alarms have to be avoided, then the larger range (304,331) will be selected, increasing the percentage of misses significantly. A more accurate analysis can be performed once the costs $C_{miss}$ and $C_{f.a.}$ of a fault miss and a false alarm, respectively, are estimated. We can then minimize the expected cost per pixel, $C_p$, expressed by

$C_p = p_f \, C_{miss} \, P_{miss} \; + \; (1 - p_f) \, C_{f.a.} \, P_{f.a.}$

where $p_f$, $P_{miss}$, $P_{f.a.}$ are the probabilities of a faulty pixel, miss and false alarm, respectively.

## 5.2. The Spatial Locality Filter

Observing Fig. 5, we see that the Natural Bounds filter by itself is not sufficient since it is unable to reduce both misses and false alarms at the same time. By adding the Spatial Locality filter, we may detect faults which have been missed by the Natural Bounds filter. Some of the detected faults will be the same, since the two filters are not totally uncorrelated – a very high pixel value will usually also result in a high difference between it and its neighbor. Fig. 5 shows that the Spatial Locality filter is better in reducing both misses and false alarms. This is a result of the dataset and fault model - the errors in pixels are in general larger than the differences between adjacent pixels. For smaller errors (or for a less "stable" dataset) we may expect the Natural Bounds filter to be better.

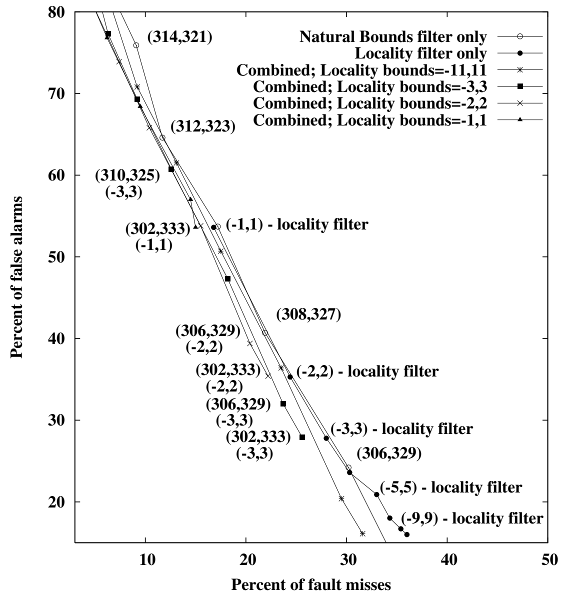


**Figure 5. Percentages of false alarms and fault misses for various bounds of the individual and combined filters**

## 5.3. The Union of Both Filters

Better fault detection can be expected when the two filters are used in series, i.e., a pixel is considered faulty if it fails any of the two filters. Fig. 5 shows the false-alarm probability ($P_{f.a.}$) vs. the miss probability ($P_{miss}$) trade-offs for several configurations of both filters' bounds (the curves marked "Combined").

Although the two filters' effects are not cumulative, we can achieve better fault detection with fewer false alarms by using the union of the two filters. By using two filters instead of one, we are making it more difficult for a pixel to be accepted as correct and thus both the fault detection and the false alarm probabilities are increased. Still, as evident from Fig. 5, the increase in fault detection rate is higher than the increase in false alarm rate, and thus, lower (i.e., better) curves of $P_{f.a.}$ vs. $P_{miss}$ can be obtained.

The best configurations of the two filters' bounds are the points which are labeled in Fig. 5. As mentioned before, the configuration to be selected out of these points depends on the relative costs $C_{miss}$ and $C_{f.a.}$ of the two possible errors, and is the one which minimizes the average pixel cost $C_p$ defined above.

Numerical examples of the fault coverage possible with these filters are available in [9].

## 5.4. Other Datasets

All the results described so far are based on the "Blob" dataset. Applying the same filter configuration to the "Stripe" set, yields very similar results. If the filters are applied to the "Spots" dataset, the Natural Bounds filter results in worse misses vs. false-alarm combinations, since the most frequent data values for the "Spots" are lower than those in the other two datasets. This is a shortcoming of the simple absolute Natural Bounds filter - any deviation in range will result in many incorrect identifications. This is another reason why an application should not rely on one filter and finding filters which are strongly disjoint would be beneficial.

### 5.5. Correction Error

The trade-off between time investment and calculation quality is a variable for the end user to decide upon, which ALFTD supports by being scalable.

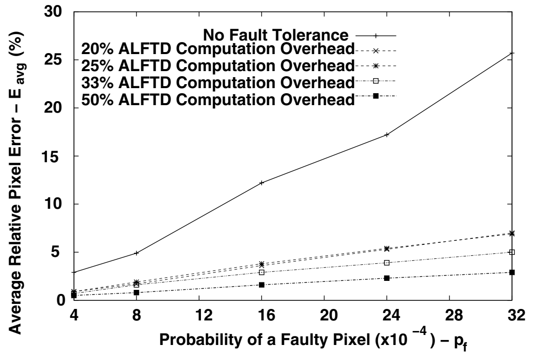


**Figure 6. Average ALFTD correction error as a function of fault probability, for several calculation overheads**

Fig. 6 shows the relationship between investment in secondary calculation runtime and the incurred relative error averaged over the entire dataset as a function of the pixel error probability. The 50% granularity provides the best improvement over the no correction case, but even the 25% and 20% granularities produce a reasonably small error.

Since OTIS's output can also be represented as a greyscale bitmap, we used a graphical representation to demonstrate the trade-off between calculation overhead and fault correction. Fig. 2 shows series of temperature outputs which were collected by injecting faults into the center rows, and then corrected with ALFTD. In these pictures, the effects of even small uncorrected errors are visible, resembling random snow. The diagrams show the effect of ALFTD on output. While the 25% output is numerically reasonable, the image is disappointing (but passable). The 50% output is good, but comes with a heavy overhead. The compromise, 33%, seems to be the best choice.

## 6. Conclusion

ALFTD provides fault-tolerance with a significantly smaller overhead than that of a full replication. It can be combined with other methods of fault tolerance to increase existing coverage.

The advantage of ALFTD comes from being implemented at the application level. The overall cost of employing ALFTD is relative to the frequency of faults, the quality of the filtering bands, and the tightness the user wants to see in their data bounds. Our results indicate that ALFTD is a suitable method for fault tolerance and detection in OTIS. As ALFTD is applied to a broader spectrum of applications, new methods for fault filtering and secondary scaling will arise, allowing it to perform with lower overhead and higher accuracy.

**Acknowledgment:** The authors wish to thank Bill Gustafson of the University of Washington for his help by both providing, and assisting us to better understand the OTIS application.
This paper was supported in part by the National Science Foundation under grant CCR-0104482.

## References

[1] R. Ferraro, "Remote exploration and experimentation project plan," Tech. Rep., Jet Propulsion Lab, July 2000.

[2] K. Huang, J.A. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Trans. on Computers*, vol C-33, pp. 518-528, Jun. 1984.

[3] P. Banerjee et al, "Algorithm Based Fault Tolerance on Hypercube Multiprocessors,"IEEE Trans. Computers, Vol. 39, No.9, pp 1132-1142, Sep. 1990.

[4] D.L. Tao, C.R.P Hartmann, and Yunghsing S. Han, "New Encoding/Decoding Methods for Designing Fault-Tolerant Matrix Operations," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, pp. 931-838, Sept. 1996.

[5] S. Yajnik and N. K. Jha, "Graceful Degradation in Algorithm-Based Fault Tolerance Multiprocessor Systems," *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, pp. 137-153, Feb. 1997.

[6] J. Haines, V. Lakamraju, I. Koren, and C.M. Krishna, "Application-level fault tolerance as a complement to system- level fault tolerance," *The Journal of Supercomputing*, vol. 16, pp. 53–68, 2000.

[7] B. Van Voorst, L. Pires, R. Jha, and M. Muhammed, "Implementation and results of hypothesis testing from the c3i parallel benchmark suite," *Proc. of the 11th Intern. Parallel Processing Symposium*, 1997, pp. 192-196.

[8] Message Passing Interface Forum MPIF, "MPI: A message-passing interface standard," Tech. Rep. UT-CS-94-230, Message Passing Interface Forum, 1994.

[9] E. Ciocca, I. Koren, and C.M. Krishna, "Determining acceptance tests for application-level fault detection," *Proc. of the 2nd ASPLOS EASY Workshop,* Oct. 2002, pp. 47–53.