, , 1–16 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Application-Level Fault Tolerance as a Complement to System-Level Fault Tolerance

JOSHUA HAINES jhaines@ecs.umass.edu
VIJAY LAKAMRAJU vlakamra@ecs.umass.edu
ISRAEL KOREN koren@ecs.umass.edu
C. MANI KRISHNA krishna@ecs.umass.edu
Electrical and Computer Engineering Dept., University of Massachusetts, Amherst, MA 01003

Abstract. As multiprocessor systems become more complex, their reliability will need to increase as well. In this paper we propose a novel technique which is applicable to a wide variety of distributed real-time systems, especially those exhibiting data parallelism. System-level fault tolerance involves reliability techniques incorporated within the system hardware and software whereas application-level fault tolerance involves reliability techniques incorporated within the application software. We assert that, for high reliability, a combination of system-level fault tolerance and application-level fault tolerance works best. In many systems, application-level fault tolerance can be used to bridge the gap when system-level fault tolerance alone does not provide the required reliability. We exemplify this with the RTHT target tracking benchmark and the ABF beamforming benchmark.

Keywords: distributed real-time systems, fault tolerance, checkpointing, imprecise computation, target tracking, beam forming.

1. Introduction

In a large distributed real-time system, there is a high likelihood that at any given time, some part of the system will exhibit faulty behavior. The ability to tolerate this behavior must be an integral part of a real-time system. Associated with every real-time application task is a deadline by which all calculations must be completed. In order to ensure that deadlines are met, even in the presence of failures, fault tolerance must be employed. In this paper we consider fault tolerance at two separate levels, system-level and application-level.

System-Level Fault Tolerance encompasses redundancy and recovery actions within the system hardware and software. While system hardware includes the computing elements and I/0 (network) sub-system, the system software includes the operating system and components such as the scheduling and allocation algorithms, checkpointing, fault detection and recovery algorithms. For example, in the event of a failed processing unit, the component of the system responsible for fault tolerance would take care of rescheduling the task(s) which had been executing on the faulty node, and restarting them on a good node from the previous checkpoint.

Application-Level Fault Tolerance encompasses redundancy and recovery actions within the application software. Here various tasks of the application may communicate in order to learn of faults and then provide recovery services, making use of some data-redundancy. In certain situations, we find that fault tolerance at the application-level can greatly augment the overall fault-tolerance of the system. For example, if a task's checkpoint is very large, application-level fault tolerance can help mask a fault while the system is moving the large checkpoint and restarting the task on another node.

N-Modular Redundancy is a well-known fault tolerance technique. A number of identical copies of the software are run on separate machines, the output from all of them is compared, and the majority decision is used [1]. This technique however, involves a large amount of redundancy and is thus costly.

The recovery block approach combines elements of checkpointing and backup alternatives to provide recovery from hard failures [2]. All tasks are replicated but only a single copy of each task is active at any time. If a computer hosting an active copy of a task fails, the backup is executed. The task may be completely restarted (which increases the chances of a deadline miss) or else executed from its most recent checkpoint [4]. The later option requires that the active copy of the task periodically copy (checkpoint) its state to its backups. This can entail a large amount of overhead, especially when the state information to be transferred is large. Such is the case with the applications that we are dealing with.

Another common technique is the use of less precise (i.e., approximate) results [3], obtained by operating on a much smaller data set, using the same algorithm. A data set can be chosen such that a sufficiently accurate result can be obtained with a greatly reduced execution time. A smaller data set is chosen either by prioritizing the data set or by reducing the granularity. Examples of such applications are target tracking and image processing, where it is better to have less precise results on time, rather than precise results too late or not at all. Our recovery technique caters to applications that exhibit data- parallelism, involves a large data set and can make do with a less precise result for a short period of time.

Our approach makes use of facets of the recovery block technique and employs reduced precision state information and results in order to tolerate faults. We employ a certain degree of redundancy within each of the parallel processes. The application as a whole is able to make use of that redundancy in the event of a fault to ensure that the required level of reliability is achieved. We consider only failures that render a process' results erroneous or inaccessible. In the case of such a fault, the redundant element's less precise results are used instead of those from the failed process. In this way, our technique can provide a high degree of reliability with only a small computational overhead in certain applications.

Section 2 introduces the RTHT and ABF benchmarks that will be used to demonstrate our technique. In Section 3 we describe in detail our application-level fault tolerance technique. Section 4 analyzes the effectiveness of this technique when used in conjunction with each of the benchmarks, and Section 5 concludes the paper.

2. The Benchmarks

Each of the benchmarks has the form shown in Figure 1. There are multiple, parallel *application* processes, which are fed with input data from a source - in this case, a *source* process which simulates a radar system or an array of sonar sensors. When the parallel computations are complete, the results are output to a *sink* process, simulating system display or actuators. Our technique is concerned with the ability to withstand faults at the parallel processes.

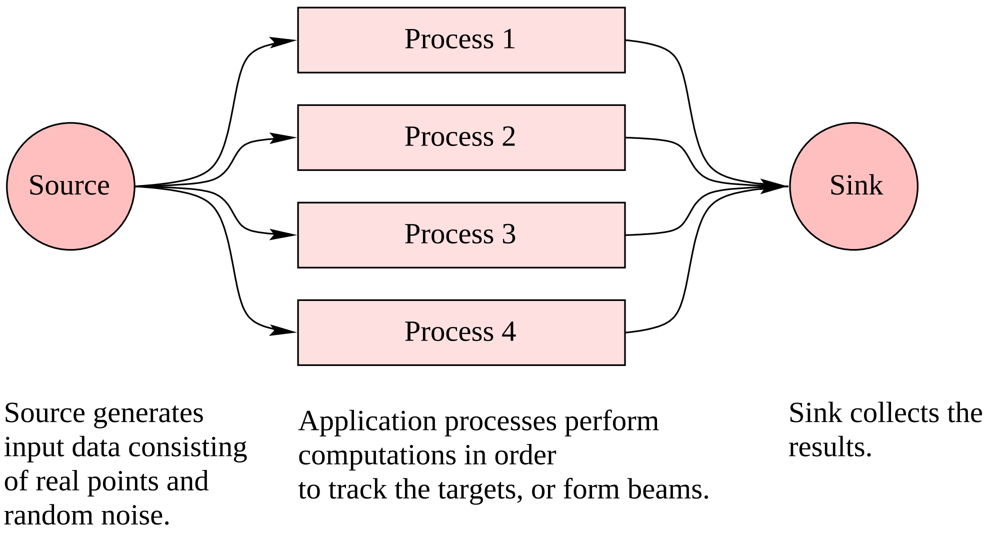
Process 1
Process 2
Process 3
Process 4
Source
Sink
Source generates input data consisting of real points and random noise.
Application processes perform computations in order to track the targets, or form beams.
Sink collects the results.

Figure 1. Software architecture of both the RTHT and ABF benchmarks.

2.1. The RTHT Target Tracking Benchmark

The Honeywell Real-Time Multi-Hypothesis Tracking (RTHT) Benchmark [6, 7], is a general-purpose, parallel, target-tracking benchmark. The purpose of this benchmark is to track a number of objects moving about in a two-dimensional coordinate plane, using data from a radar system. The data is noisy, consisting of false targets and clutter, along with the real targets. The original, non-fault-tolerant application consists of two or more processes running in parallel, each working on a distinct subset of the data from the radar. Periodically, frames of data arrive from the radar, or source process in this case, and are split among the processes for computation of hypotheses. Each possible track has an associated hypothesis which includes a figure of likelihood, representing how likely it is to be a real track. A history of the data points and a covariance matrix are used in generating up-to-date likelihood values.

For every frame of radar data, each parallel process performs the following steps: 1) Creation of new hypotheses for each new data point it receives, 2) Extension of existing hypotheses, making use of the new radar data and the existing covariance matrix, 3) Participation in system-wide compilation or ranking of hypotheses, led by a *Root* application process, and 4) Merging of its own list of hypotheses with the system-wide list that resulted from the compilation step. The deadline of one frame's calculations is the arrival of the next frame.

By evaluating the performance of the original, non-fault-tolerant, benchmark when run in conjunction with our RAPIDS real-time system simulator [9], it became apparent that despite the inherent system-level fault tolerance in the simulated system, the benchmark still saw a drastic degradation of tracking accuracy as the result of even a single faulty node. Even if the benchmark task was successfully reassigned to a good node after the fault, the chances that it had already missed a deadline were high. This was in part due to the overheads associated both with moving the large process checkpoint over the network and with restarting such a large process. Once the process had missed the deadline, it was unable to take part in the compilation phase and had to start all over again and begin building its hypotheses anew. This took time, and caused a temporary loss of tracking reliability of up to five frames. Although better than a non-fault-tolerant system, in which that process would simply have been lost, it was still not as reliable as desired.

We decided to address two points, in order to improve the performance of the benchmark in the presence of faults: 1) The overhead involved with moving such a large checkpoint and 2) A source of hypotheses for the process to start with after restart.

Our measure of reliability is the number of *real targets* successfully tracked by the application (within a sufficient degree of accuracy) as a fraction of the exact number of real targets that should have been tracked. To simplify this calculation, the number of targets is kept constant and no targets enter or leave the system during the simulation.

2.2. The ABF Beam Forming Benchmark

The Adaptive Beam Forming (ABF) Benchmark [8] is a simulation of the real-time process by which a submarine sonar system interprets the periodic data received from a linear array of sensors. In particular, the goal is to distinguish signals from noise and to precisely identify the direction from which a signal is arriving, across a specified range of frequencies. In this implementation, the application receives periodic samples of data as if from the linear sensor array. The data is generated so that it contains four reference *beams*, or signals, arriving from distinct locations in a 180-degree field of view, along with random noise.

The application itself consists of several application processes, each attempting to locate beams at a distinct subset of the specified frequency range. Frames of data for each frequency are "scattered" periodically from the source process. Output, in the form of one beam pattern per frequency, is "gathered" by the sink process. Figure 2 depicts a typical beam pattern output, shown here at frame 18, frequency 250Hz, with reference beams at -20, -60, 20 and 60 degrees.

Each application process performs calculations according to the following loop of pseudo-code, for each frame of input.

```
for_each ( frequency ) {
    Update dynamic weights.
    for_each ( direction of arrival) {
      Search for signal, blocking out interference
      from other directions and frequencies.
    }
  }
```

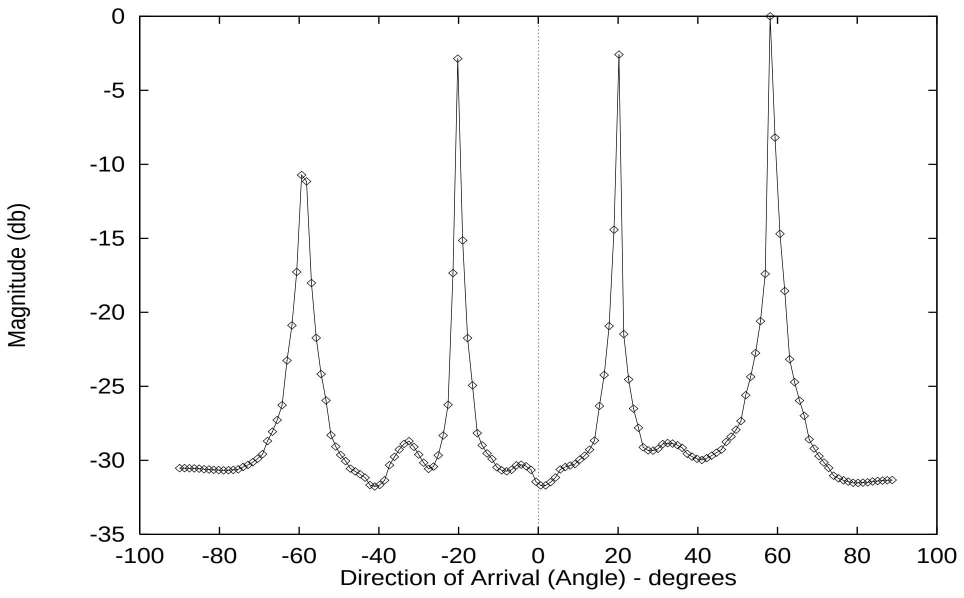

Figure 2. Typical beam pattern output.

For each frequency, the process first updates a set of weights that are dynamically modified from frame to frame. Applying these weights to the input samples has the effect of forming a beam which emphasizes the sound arriving from each direction. The process searches in each possible direction (-90 to 90 degrees) for incoming signals. The granularity of this direction is directly related to the number of sensors.

In addition, at the start of a run, there is an initialization period in which the weights are set to some initial values, and then 15 to 20 frames are necessary to “learn” precisely where the beams are.

It is evident that this sort of application faces reliability problems similar to those of the RTHT benchmark. If a processing element fails, all output for those frequencies is lost during the down time, and when the lost task is finally replaced by the system, it has to go through a startup period all over again. Here, too, the data sets of these processes are very large, creating a considerable overhead if checkpointing is employed. To avoid the delay associated with this overhead, be able to maintain full output during the fault, and provide quick restart after the fault, application-level fault tolerance must be employed.

We evaluate the quality of the ABF output with two tests applied to the resultant beam pattern. In the Placement Test we check whether the direction of arrival of the beam has been detected within a certain tolerance. In the Width Test the aim is to determine how accurately the beam has been detected by measuring the width of the beam, in degrees, at 3db down from the peak. A beam that passes both tests is considered to be correctly detected.

3. Implementation of Application-Level Fault Tolerance

Our technique uses redundancy in the form of extra work done by each process of the application. Each process takes, in addition to its own distinct workload, some portion of its *neighbor*'s workload, as shown in Figure 3. The process then tracks beams or targets for both its own work and overlaps part of its neighbor's, but makes use of the redundant information only in case this neighbor becomes faulty. We now explain briefly how the data set is divided, how the application might learn of faults, and how it would recover from them.

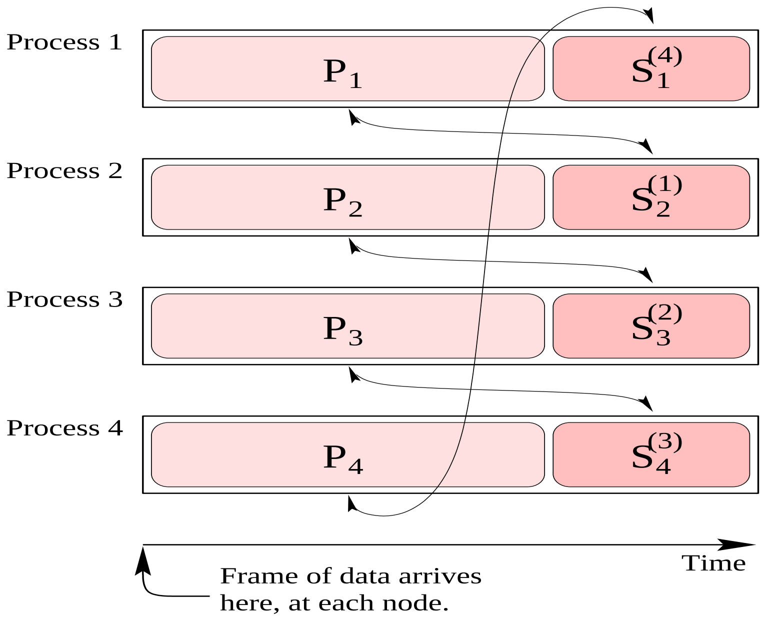

Figure 3. Architecture of both benchmarks with application-level fault tolerance.

3.1. Division of Load

The extent of duplication between two neighboring nodes will greatly affect the level of reliability which can be achieved. Duplication arises from the way we divide the data set among the parallel processing nodes. First, each frame of data is divided as evenly as possible among the nodes. The section of the process that takes on this set of data is the primary task section, P_i. Then we assign each node, n_i, some additional work: part of its neighbor, n_{i-1}'s, primary task. The section of the process that takes on this set of data is the secondary task section, S_i. In other words,

- The *primary task section*, P_i, refers to the calculations which node n_i carries as part of the original application.
- The *secondary task section*, S_i, refers to the calculations which node n_i carries out as a backup for its neighbor, n_{i-1}. Node n_1 hosts the secondary corresponding to the primary running on the highest numbered node. The secondary section, S_i, will be kept in synchronization with the primary P_{i-1}.

3.2. Detection of Faults

There are two ways in which fault detection information can reach the various application processes. In the first, the system informs the application of a faulty node, and the second is through specific timeouts at the phase of the application where communication is expected. The former would typically incur the cost of periodic polling, while the latter could result in late detection of the fault. Although the exact integration of application-level fault tolerance would vary depending on the fault detection technique chosen, the effectiveness of our technique should not.

3.3. Fault Recovery

If, at a deadline prior to that of the frame, node n_i is discovered to be faulty and is unable to output any results, then node n_{i+1} which is serving as its backup will send as output S_{i+1}'s data in place of the data that n_i is unable to supply. In the meantime, the system will be working on replacing or restarting the process that was interrupted by the fault. In fact, the system's job here is made easier by the fact that if the process has to be restarted on another node, the process data segment no longer needs to be moved. When the process is rescheduled, it will make use of the information maintained by its secondary on its behalf in order to pick up where it left off before the fault. This way, the application fault tolerance is able to work in conjunction with the system fault tolerance. This will help even in the case of transient faults, in that the application-level fault tolerance allows more leeway to postpone the restarting of the process on another node, in the hope that the fault will soon disappear.

3.4. Extension to a higher level of redundancy

Our technique guarantees the required reliability in the presence of one fault but could also withstand two or more simultaneous failures depending on which nodes are hit by the faults. For example, in a six-node system if the nodes running processes 1, 3, and 5 fail, the technique would still be able to achieve the required reliability. Of course, this is contingent on the assumption that the processes on the faulty nodes are transferred to a safe node and restarted by the beginning of the next frame.

3.5. Benchmark Integration Specifics

We next discuss specific details regarding the application of our technique to each of the benchmarks.

3.5.1. RTHT benchmark In the RTHT Benchmark, the "unit of redundancy" is the hypothesis. That is, each secondary task section creates and extends some fraction of the total number of hypotheses created and extended by the process for which it is secondary. The amount of secondary redundancy is expressed as a percentage of the number of hypotheses extended by the primary.

Redundancy is implemented in the following way: At the beginning of each frame, the source process broadcasts the input radar data, and hypotheses are created and extended as before, with the exception that additionally the secondary extends a percentage of those extended by the corresponding primary. The secondary section S_i is kept in synchronization with primary P_{i-1} via the compilation process, which in this case is again a process-level broadcast communication, so that no extra communication is necessary. If node n_i is discovered to be faulty and is unable to participate in the compilation of that frame, then node n_{i+1} which is serving as its backup will make use of S_{i+1}'s data in the compilation process in place of the data that n_i is unable to supply.

When the process is rescheduled, it will make use of the hypotheses extended by the secondary on its behalf so as to pick up where it left off. This information is obtained from the secondary process by way of compilation - the newly rescheduled process merely listens in on the compilation process and copies those hypotheses which have been extended by its secondary.

3.5.2. ABF benchmark There are two ways in which we have integrated application-level fault tolerance with the ABF Benchmark. They differ in the manner in which the secondary abbreviates the calculations of the primary so as to obtain a full set of results. The methods are:

- The Limited Field of View (Limited FOV) Method in which the secondary looks for beams at every frequency as in the primary, however it searches only a subsection of the primary's field of view (divided into one or more segments). Ideally the secondary will place these "windows" at directions in which beams are known to be arriving. We impose a minimum width of these windows, due to the fact that if an individual window is too narrow, the output could always (perhaps erroneously) pass the width-based quality test, described in section 2.

The amount of redundancy is expressed as the percentage of the field of view searched by the secondary.

- The Reduced Directional Granularity Method in which the secondary looks for beams at every frequency and in every direction, but with a reduced granularity of direction. The amount of redundancy is expressed as a percentage of the original granularity computed by the primary.

Both techniques serve to reduce the computational time of the secondary task set, while maintaining useful system output. In addition, the two techniques may be employed concurrently in order to further reduce the computational time required by the secondary task.

To implement either variation of the technique, the input frame of data is scattered a second time from the source to the application processes. This is time - rotated, so that each process receives the input data of the process for which it is a secondary. Each process first carries out its primary computational tasks, and then carries out its secondary task. At the frame's deadline, if a process is detected to be down, the sink will gather output from the non-faulty processes, including the backup results from the process that is secondary to the one that is faulty. In the event of an application process being restarted after a fault, it will receive the current set of weights from its secondary in order to jump-start its calculations.

Some synchronization between primary and secondary is required in the Limited FOV Method. It is a small, periodic communication in which either the sink process or the primary itself tells the secondary at what frequencies and directions it is detecting beams. Such synchronization is not necessary for the Reduced Granularity Method.

4. Results

4.1. The RTHT Benchmark

When applied to the RTHT benchmark, we found that only a small amount of redundancy between the primary and secondary sections is necessary in order to provide a considerable amount of fault tolerance. Furthermore, the increase in system resource requirements, even after including overheads of the technique's implementation, is minimal compared to that of other techniques, in achieving the same amount of reliability. These points are demonstrated in Figures 4, 5, and 6. Each run contains 30 targets which remain in the system until the end of the simulation (the 30th frame), as well as some number of false alarms. The case when only system-level fault tolerance exists corresponds to the case when the secondary extends 0% of the primary hypotheses.

In Figure 4 we see the number of targets which are successfully tracked, when we have just two application processes and a fault occurs at frame 15. (In this case there were roughly 80 false alarms per frame of data.) In this run, 15% redundancy allows us to track all of the real targets, despite the fault. We can attribute the fact that a small amount of redundancy can have a great effect on the tracking stability,

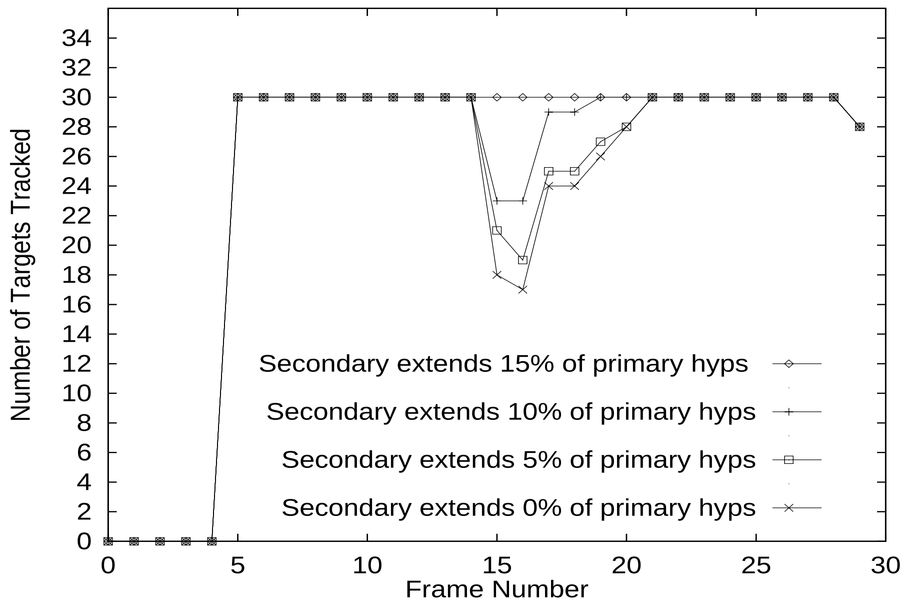

Figure 4. Tracking accuracy, in number of real targets tracked for a given percentage of redundancy.

to the fact that the hypotheses which are being extended by the secondary are the ones *most likely* to be real targets. At the beginning of the compilation phase, each application process sorts its hypotheses, placing the *most likely* at the head of the list for compilation. Thus, at the beginning of the next frame, each application process and its secondary begin extending those hypotheses with the highest chance of being real targets.

To refine this point, Figure 5 shows the average percentage of redundancy required for a given number of application processors and a single fault, as before. The amount required shows a gradual decrease as we add more processors. We can attribute this to the fact that the chance of a single process containing a high percentage of the real targets decreases as processors are added.

In addition, a proportionately small load is imposed on the processor by the computation of the secondary task set, as seen in Figure 6. This can be attributed to the fact that a hypothesis whose position and velocity are known precisely, does not take as much time to extend compared to those hypotheses which are less well-known. And since the *most likely* hypotheses are generally the most well-known *and* are the hypotheses which the secondary extends, the amount of processor time taken to execute the secondary task is proportionally much smaller.

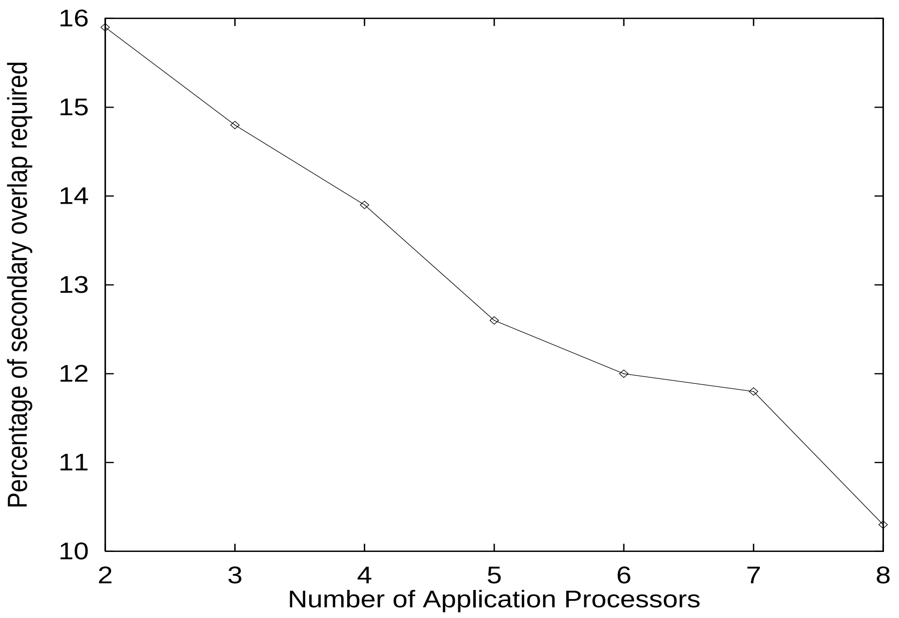

Figure 5. Average minimum percentage of secondary overlap required to miss no targets despite one node being faulty.

4.2. ABF Benchmark Results

When we integrate application-level fault tolerance with the ABF benchmark, we find that only a small amount of redundancy is necessary to ensure complete masking of single frame faults. With either variation (reduced granularity or limited FOV method) we see that a secondary redundancy of 33% is adequate to provide complete and accurate results in the faulty frame and the following frames (after the faulty process is restarted). If we combine the two techniques, we see an even further reduction in the computational effort imposed by the secondary in order to mask the fault. We have not taken additional network overhead and/or latency into account in figures of overhead - they refer solely to computational overhead. Network overhead will depend greatly on the medium used. In particular, a shared medium would allow the secondary to “snoop” on the primary’s input and output, eliminating the need for additional communication.

All results were obtained by running simulations with 75 sensors and four reference input beams for 50 frames. There are two application processors, and a fault occurs in one of them at frame 30. Results are presented and discussed for three redundancy methods: the Limited FOV method, the Reduced Granularity method and a Combined method (a combination of the first two). The quality of the results is assessed by totalling the number of beams that were tracked successfully. Here, there are four input beams at each frequency and 32 frequencies – making 128

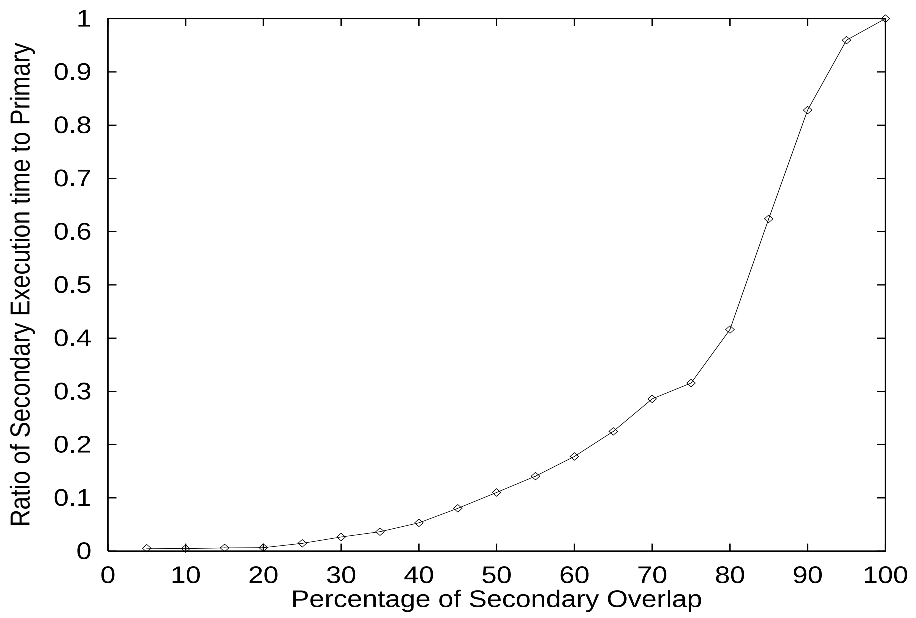

Figure 6. Ratio of time taken to compute the secondary hypothesis to the time to compute the primary hypothesis versus the percentage of secondary overlap.

beams in all. As an example, Figure 7 presents the results for several runs of the ABF benchmark while utilizing the Limited FOV redundancy method alone, with a single processor fault occurring at frame 30 and lasting one frame. We see that a 30% overlap is adequate to preserve all beam information within the system despite the loss of one processor in frame 30. We have tabulated the results for all three methods in Table 1.

4.2.1. ABF Results: Limited FOV Alone As we see in Table 1, roughly 30% secondary overlap is adequate to provide full masking of the fault. The computational overhead imposed by the secondary is about 30%. In addition, Figure 8 shows the rather linear increase in overhead as we increase the fraction of overlap.

Table 1. Amount of secondary overhead imposed by various redundancy methods, each of which is capable of fully masking a single fault.

Redundancy Technique	Secondary Overlap	Computational Overhead
Reduced Granularity	33%	35%
Limited FOV	30%	30%
Combined - 30%FOV,50%Granularity	15%	17%

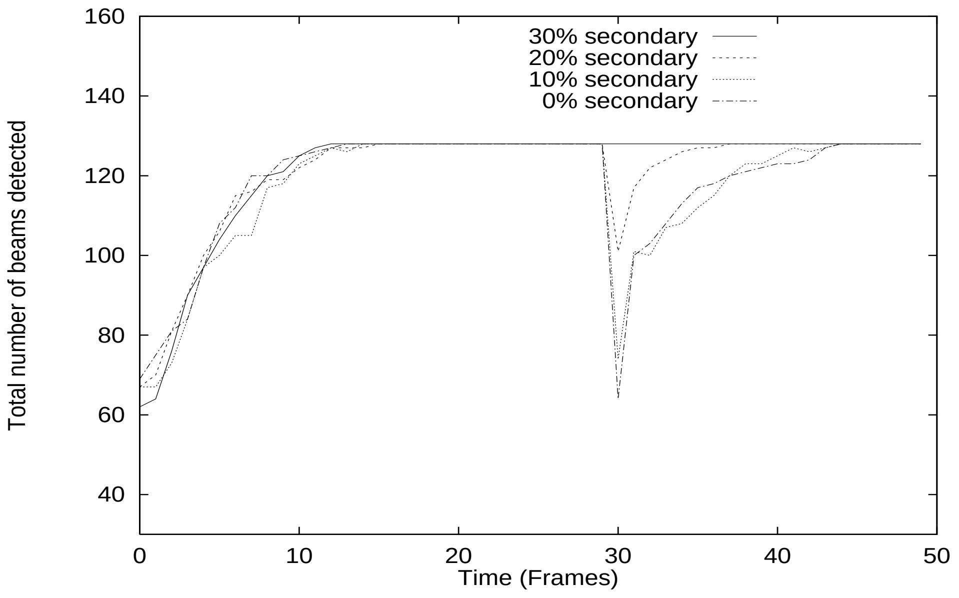

Figure 7. The number of beams correctly tracked in each frame, for the given levels of redundancy, for the Limited Field of View Method. A single process experiences a fault of duration one frame, at frame 30.

Associated with this technique however, is a potential dependence on the number of beams detected in the system, as described earlier. In order to ensure that the width test applied to the output can fail, we impose a minimum window-width. This minimum width dictates that for a given amount of overlap, there is a maximum number of windows in which the secondary may search for beams. If there are more beams than the maximum number of windows then some may be missed by the secondary search, depending on the direction of arrival. However, the system designer can lessen the likelihood of this occurring by carefully choosing the amount of overlap allotted, and tuning the criteria with which areas will be searched by the secondary.

4.2.2. ABF Results: Reduced Granularity Alone Here, too, we see that, according to Table 1, operating the secondary at 33% of the granularity of the primary results in complete masking of the fault, and that this imposes a 35% overhead to the processing node. Figure 8 again shows a linear relationship between the computational overhead and the overlap, and indicates that the overhead of the method itself is a bit higher than that of the Limited FOV method. When considering the Reduced Granularity method, we see no dependence on the number of beams detected, although beams could be missed if their peaks were within a few degrees of each other, and the granularity were very coarse.

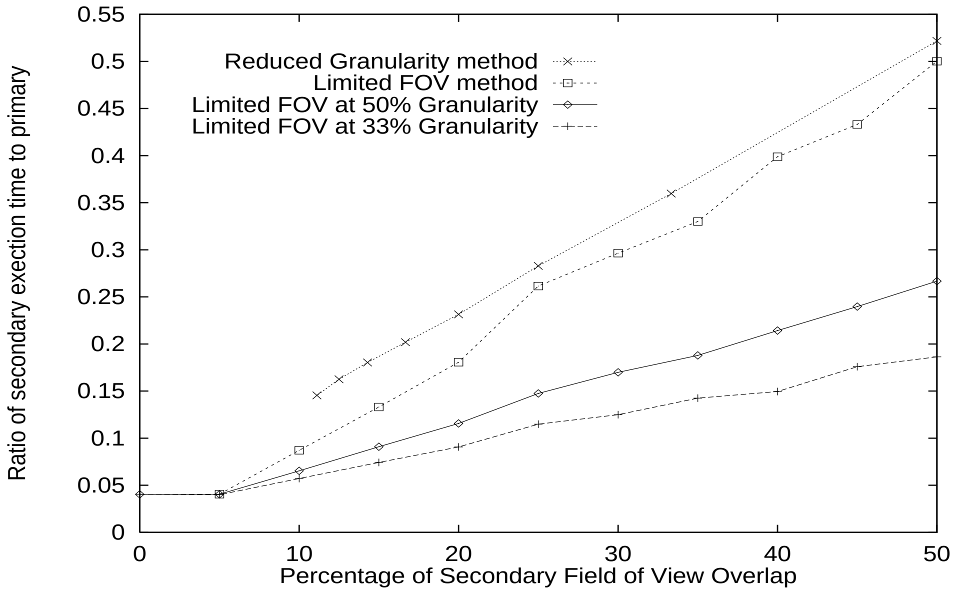

Figure 8. The ratio of secondary to primary execution time for the variations of application-level fault tolerance integrated with the ABF Benchmark versus the percentage of secondary field of view overlap.

4.2.3. ABF Results: Combined methods When we combine these two techniques, we see the greatest reduction in computational overhead of the secondary task. As shown in Table 1, a 30% field of view combined with a 50% granularity maintains the tracking ability similar to that of either one alone, yet cuts the computational overhead nearly in half. This reduction is illustrated in Figure 8, in the lower two curves, representing the overhead imposed as we vary the field of view and make use of 50% and 33% granularity respectively.

5. Conclusions

A high degree of fault tolerance may be obtained with a minimal investment of system resources in applications exhibiting data parallelism, such as the ABF and RTHT Benchmarks. It is achieved through a combination of application-level and system-level fault tolerance. A prioritized ordering within the data set, as in the RTHT benchmark, or a reduced granularity, as in the ABF benchmark, is made use of, to decrease the computational overhead of our technique.

The processes in these benchmarks are very large, so that moving a checkpoint and restarting the task may take a significant amount of time. The application-level fault tolerance is able to ensure that, despite the temporary loss of the task, the required reliability is maintained.

Since the primary and secondary task sets are incorporated within a single application process, the primary is always executed first and the secondary next. Once the primary has completed, it may alert the scheduler, indicating that the secondary need not be executed. It is useful, but not necessary, for the secondary to still be executed, as this allows it to be better synchronized with its primary counterpart. If a fault is detected, the priority of the secondary could be raised, to ensure that it will complete without missing its deadline, and provide the necessary data for compilation.

This technique is a substantial improvement over complete system duplication, in that it does not require 100% system redundancy, but merely adds a small amount of load to the existing system in achieving the same amount of fault tolerance. It differs from the recovery block approach in that the secondary does not have to be cold-started, but is ready for execution when a failure of the primary is detected. In addition, the level of reliability may be varied by varying the amount of redundancy.

In order to integrate such application-level fault tolerance, the designer will need to first determine how to prioritize the data set and/or reduce the granularity in order to define the secondary's dataset. Second, the designer should choose mechanisms by which the secondary gets the input data it needs, is able to output results when necessary, and is able to communicate with the primary for synchronization purposes. Naturally, some sort of fault detection will have to used as well. The designer must carefully weigh the overheads imposed by various methods to achieve fault tolerance and the quality of results that may be obtained from each.

In conclusion, we believe that steps to integrate this technique into the application should be taken right from the early stages of the design in order for this approach to be most effective.

Acknowledgments

This effort was supported in part by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0341, order E349. The government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Air Force Research Laboratory, or the U. S. Government.

References

1. D.P. Siewiorek and R.S. Swarz *Reliable Computer Systems Design and Evaluation*, 2nd ed. Digital Press, Burlington, MA, 1992.
2. B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, vol. SE-1, pp. 220-232, 1975.
3. J.W.S. Liu, W. Shih, K. Lin, R. Bettati, and J. Chung. Imprecise Computations. *Proceedings of the IEEE*, vol. 82, No. 1, pp. 83-93, Jan. 1994.

4. N.A. Speirs and P.A. Barrett. Using Passive replicates in Delta-4 to Provide Dependable Distributed Computing. *Proceedings of the Nineteenth International Symposium on Fault-Tolerant Computing*, 1989, pp. 184-190.
5. A.L. Liestman and R.H. Campbell. A Fault-Tolerant Scheduling Problem. *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 1089-1095, Nov. 1986.
6. B. VanVoorst, R. Jha, L. Pires, M. Muhammad. Implementation and Results of Hypothesis Testing from the C^3I Parallel Benchmark Suite. *Proceedings of the 11th International Parallel Processing Symposium*, 1997.
7. D.A. Castanon and R. Jha. Multi-Hypothesis Tracking (Draft). DARPA Real-Time Benchmarks, Technical Information Report (A006), 1997.
8. R. Hamza, Honeywell Technology Center. Sonar Adaptive Beamformer (Draft). DARPA Real-Time Benchmarks, Primary Technical Information Report, 1998.
9. M. Allalouf, J. Chang, G. Durairaj, V.R. Lakamraju, O.S. Unsal, I. Koren, C.M. Krishna. RAPIDS: A Simulator Testbed for Distributed Real-Time Systems. *Advanced Simulation and Technology Conference*, 1998, pp. 191-196.
10. C.M. Krishna and K.G. Shin *Real-Time Systems*, McGraw Hill, New York, NY, 1997.