DEVELOPMENT OF APPLICATION-LEVEL FAULT TOLERANCE IN A REAL-TIME BENCHMARK

Josh Haines, Vijay Lakamraju, Israel Koren and C. M. Krishna

Department of Electrical and Computer Engineering
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. 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.

1 Introduction

Associated with every real-time application task is a deadline by which all calculations must be complete, and the result(s) must be output. In order to ensure that deadlines are met, even in the presence of failure, fault tolerance must be employed. Here we deal with fault tolerance at two levels, system-level and application-level.

- *System-Level Fault Tolerance:* This encompasses all redundancy of system components and recovery actions taken by the system. The components involved might include operating systems, scheduling/allocation algorithms, redundant hardware/network configurations, 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) at the faulty node, and restarting them on a good node from the previous checkpoint.
- *Application-Level Fault Tolerance:* 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.

We present a solution whereby there is a degree of redundancy within each of the parallel application processes, and 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 guaranteed.

In Section 2 the reader is introduced to the RTHT benchmark, and in Section 3 we describe our application-level fault tolerance technique. In Section 4 we analyze the effectiveness of this technique when used in conjunction with the RTHT benchmark. Section 5 concludes the paper.

2 The RTHT Benchmark

The Honeywell Real-Time Multi-Hypothesis Tracking (RTHT) Benchmark [2, 3], is a general-purpose, parallel, target-tracking benchmark. The purpose of this benchmark is to track a number of objects moving about in a 2-dimensional coordinate plane, using data from a radar system. The data is noisy, with lots of false alarms, consisting of false-targets and clutter. 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 and are split between 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, lead by a *Root* application process, and 4) Merging 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 [4], 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 would not be able to take part in compilation and would have 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 5 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.

3 Implementation of Application-Level Fault Tolerance

The 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 set of radar data, some portion of its *neighbor*'s data set. The process then creates and extends hypotheses for both its own data and part of its neighbor's, but makes use of the redundant information only in the case that this neighbor is hit by a fault. We explain briefly how the data set is to be divided, how the application might learn of faults, and how it would recover from them.

3.1 Division of Load

The extent of duplication between two neighboring processes will greatly affect the amount of reliability which can be achieved. Duplication arises from the way we divide the data set among the parallel processes. First, each frame of data is divided as evenly as possible among the application processes. The section of the process π_i that takes on this set of data is the primary task section, P_i. Then we assign each process some additional work: part of its neighbor, π_{i-1}'s primary dataset. 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 process π_i carries as part of the original application. Hypotheses that are created out of P_i are considered to be owned by process π_i, and will be extended by that process in later frames.
- The *secondary task section*, S_i, refers to the calculations which process π_i carries out as a backup for its neighbor, π_{i-1}. The secondary section S_i will be kept in synchronization with the primary P_{i-1} via the compilation process.

3.2 Detection of Faults

There are two ways in which fault detection information can reach the various application processes. The first is that the system informs the application of a faulty node, and the second is through specific timeouts at the compilation 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.

3.3 Fault Recovery

If, at compilation time, process π_i is discovered to be unable to participate in that frame's compilation, then the process π_{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 π_i is not able to supply. In the meantime, the system will be working on replacing the process that was interrupted by the fault. In fact, the system's job here is made easier by the fact that it no longer needs to move the whole data segment of the process. When the process is rescheduled, it will make use of the hypotheses extended by its secondary on its behalf in order to pick up where it left off before the fault. This way, the application's 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 hope that the fault will 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, 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.

4 Results

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 great 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 fault tolerance. These points are demonstrated in Figures 1, 2 and 3. 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.

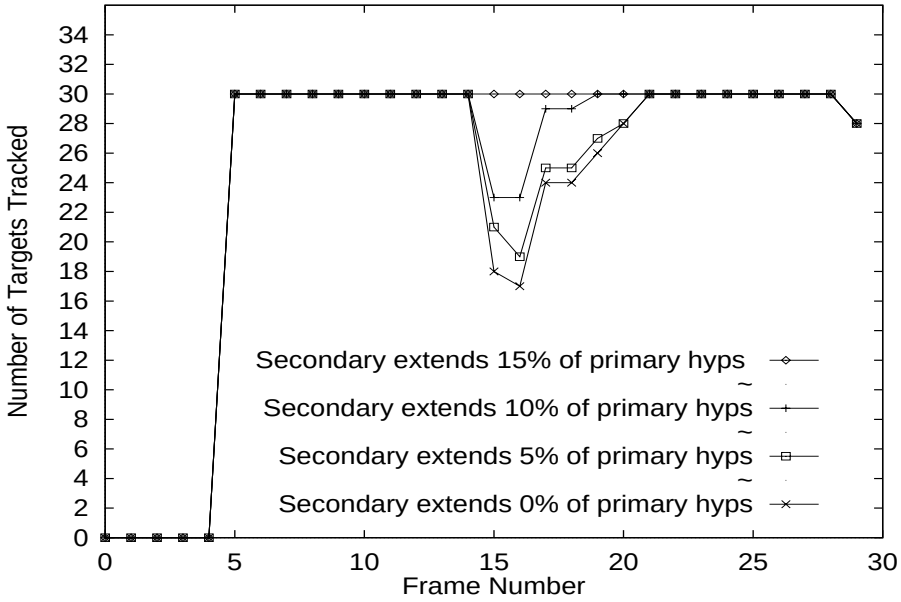

Figure 1: Tracking accuracy, in number of real targets tracked for a given percentage of redundancy.

In Figure 1, 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, 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 their secondaries begin extending those hypotheses which have the highest chance of being real targets.

To refine this point, Figure 2 shows the average percentage of redundancy required for a given number of application processes and a single fault, as before. We can attribute the fact that the amount required shows a gradual decrease as we add more processes, to the fact that the chances of a single process containing a high percentage of the targets decreases.

In addition, a proportionally small load is imposed on the processor by the computation of the secondary task set, as seen in Figure 3. This can be attributed to the fact that the extension of a hypothesis whose position and velocity are known precisely, does not take as much time to extend

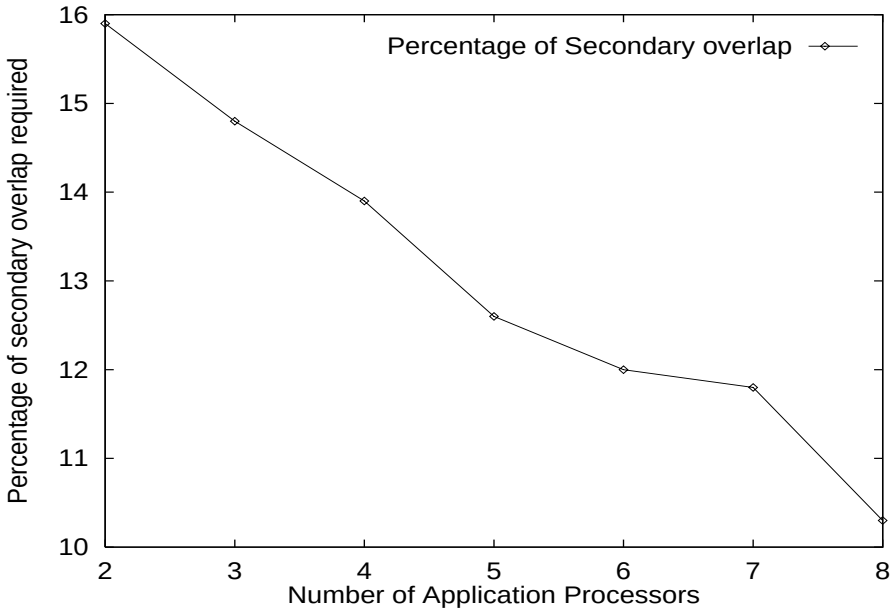

Figure 2: Average minimum percentage of secondary overlap required to miss no targets despite a fault at one node.

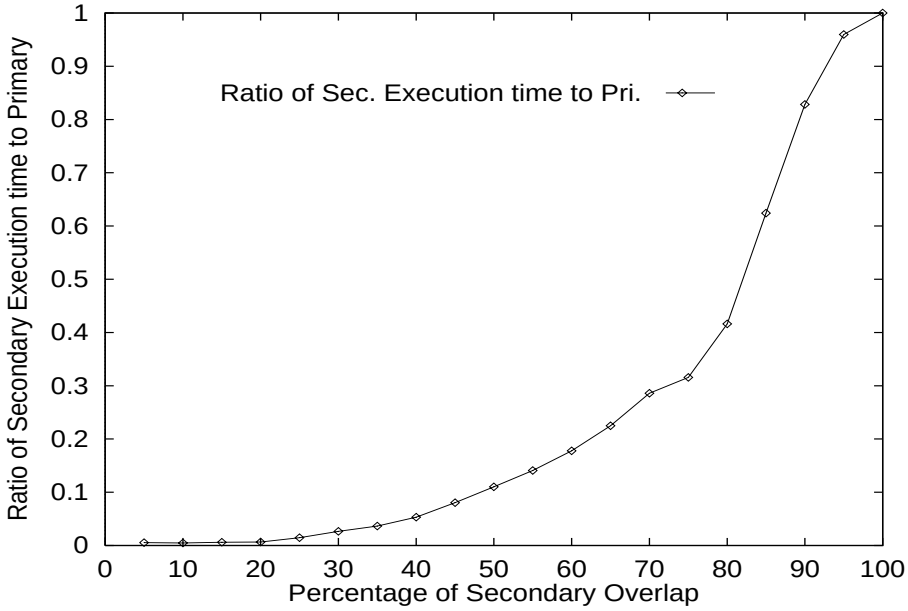

Figure 3: Ratio of time taken to compute the secondary hypothesis to the time to compute the primary hypothesis versus the percentage of secondary overlap.

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.

5 Conclusions

A high degree of fault tolerance may be obtained with a minimal investment of system resources in applications exhibiting data parallelism and a prioritized ordering within the data set. It is achieved through a combination of application-level and system-level fault tolerance.

In the case of this benchmark, the processes are very large, so that moving the checkpoint and restarting the task may take a significant amount of time. Thus the application-level fault tolerance is able to ensure that, despite the temporary loss of the primary, the require 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 conclusion, we believe that steps to integrate this technique into the application should be taken right from the early stages of design in order for this approach to be most effective.

Acknowledgment

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 Projects Agency, Air Force Research Laboratory, or the U. S. Government.

References

[1] 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.

[2] B. VanVoorst, R. Jha, L. Pires, M. Muhammad, "Implementation and Results of Hypothesis Testing from the C^3I Parallel Benchmark Suite," in *Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[3] D.A. Castanon and R. Jha, "Multi-Hypothesis Tracking (Draft)", DARPA Real-Time Benchmarks, Technical Information Report (A006), 1997.

[4] 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.

[5] C.M. Krishna and K.G. Shin: *Real-Time Systems*, McGraw Hill, New York, NY, 1997.