Using Speculative Execution For Fault Tolerance in a Real-Time System

Mohamed F. Younis, Grace Tsai, Thomas J. Marlowe and Alexander D. Stoyenko

Real-Time Computing Laboratory,
Department of Computer and Information Science
New Jersey Institute of Technology, Newark, NJ 07102, USA
younis@cis.njit.edu [*]

Abstract

Achieving fault-tolerance using a primary-backup approach involves overhead of recovery such as activating the backup and propagating execution states, which may affect the timeliness properties of real-time systems. We propose a semi-passive architecture for fault-tolerance and show that speculative execution can enhance overall performance and hence shorten the recovery time in the presence of failure. The Compiler is used to detect speculative execution, to insert checkpoints and to construct the updated messages. Simulation results are reported to show the contribution of speculative execution under the proposed architecture.

1 Introduction

There has been an increase, during the past few years, in the use of distributed computer systems in complex real-time applications, such as patient monitoring, avionics, remote sensing and air-traffic control. These real-time systems must function correctly and meet timing constraints, which requires fault-tolerance capabilities.

In real-time computing, fault-tolerance is usually achieved with some sort of redundancy. Such redundancy uses replicas to supply the same functionality in case of failure. However, keeping consistency between the replica and the primary service provider incurs overhead. To achieve fault-tolerance, both the overhead and recovery time should be predictable.

Tolerance to faults, typically, can be realized in four steps [16]: error detection, diagnosis to recognize the causes, invocation of an appropriate recovery procedure, (which may require monitoring the execution progress.) and finally, software reconfiguration if necessary. This submission focuses on enabling recovery from hardware faults.

```
  ORIGINAL                   FAULT_TOLERANT

if (exp)                   Propagate updates
  code_block1              if (exp)
else                         code_block1
  code_block2              else
                             code_block2
/* meets deadline          /* may miss deadline by
on both branches */        the time of updates */
```

Figure 1: Checkpoints propagation can result in missed deadlines.

In distributed real-time systems, two schemes, *passive* and *active* replication, are commonly used to replicate servers that fail independently. Active replication relies on voting algorithms to select the most reliable responses from the servers [4, 18, 19]. This scheme tends to have larger response time due to the overhead of the voting algorithm. Passive replication [6, 7, 24], also known as *primary-backup*, relies on one or more backup units. The primary server propagates checkpoints (execution states) to the backups. If the primary fails, one of the backups will take over. The problem of this scheme is the high overhead of propagating the state and rolling back computation (backward recovery). That overhead may lead to misses of deadlines. Considering the example in *Figure* 1, assume that *code_block1* takes 20 units of time, *code_block2* takes 10 units, and the updates propagation takes 2 units. The true branch is a member of the worst-case execution path. If the *exp* is true, the program may miss its deadline due to the propagation of updates (2 units).

In [29], we use compile-time analysis to detect both safe and profitable speculative execution in real-time systems. We rely on intensive static time analysis to

[*]Tsai is also with the department of Computer Science, Fairleigh Dickinson University, Teaneck, NJ 07666, USA. Marlowe is also affiliated with the department of Mathematics and Computer Science, Seton Hall University, South Orange, NJ 07079, USA. This work is supported in part by US ONR Grant Number N00014-92-J-1367, by US-NSWC Grant N00014-93-1-1047 and N60921-93-M-1912 , by Siemens Research Collaborative Grants and by AT&T UEDP Grant 91-134. The authors are indebted to the many constructive comments made by the anonymous referees as well as by the following Real-Time Computing Lab members: B. Ghahyazi, C. Amaro, M. Harelick, and P. Sinha.

investigate the effect of rollback on the worst-case execution. We developed compiler transformations rules to fork processes to execute parts of the codes speculatively on a shadow replica. Initial values will be copied to the shadow from the primary processor, as shown in *Figure* 2. The shadow eventually will send back the results of the computation which may be committed or discarded by the primary. Using the backups in the passive replica model as a shadow for the primary server we can enhance the overall performance. The saving in execution time can be used to absorb the state update overhead without affecting the service deadline.

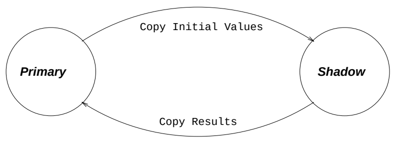

Figure 2: Speculative execution on a shadow replica

Speculative execution can be successful in computation-intensive complex systems, such as real-time imaging and multimedia. Although such applications have potential for parallelism, there are also many opportunities for speculative execution. Image filtration, for example, usually involves a lot of computation, while testing the quality of an image is time-consuming as well [5]. An image can be filtered speculatively on a shadow while quality tests are running. The same argument holds for edge detection. Moreover, morphological image processing [12] has a lot of potential for speculative execution. Construction of a structural element can be done speculatively while another element is being tried. Another application is image retrieval, according to certain input or the occurrence of an event. The most complicated image can be retrieved and filtered speculatively on a shadow to shorten the worst-case execution.

In this paper, we use a variant of the passive replica model in which some backup nodes will be semi-passive. Passive backup nodes are provided with an exact copy of process code, while semi-passive backups will have a transformed copy of the code. In ordinary serial execution, checkpoints will be propagated to all backups. The compiler will generate processes for code that can be speculatively executed on semi-passive replicas. For example, in *Figure* 1 transformed version of the code of *Figure* 1 assume *exp* involves a call that takes 30 units of time. When the primary is evaluating the condition, *block1* can simultaneously be executed on the semi-passive replica (shadow). An activation message will be sent to the shadow to activate the execution of *block1*. The code executed on the shadow will be a new process sharing the same data with the original idle process replica. The latest state can be sent to the shadow before the activation message. The shadow will send back the updated state to the primary which will be committed if *exp* is true.

The technique we are proposing can manage state update overhead and make it possible to provide support for real-time fault-tolerance. Moreover, it can shorten the time for recovery. In the above example, if the primary fails after the conclusion of the call to evaluate *exp* and *exp* happens to be true, then the semi-passive (shadow) backup can perform forward recovery (if *exp* is false, we rely on another passive backup, as shown in Section 7). In addition, it can enhance overall performance and resource utilization through the proposed redundancy.

Opportunities for speculative execution can be detected not only in conditions, but also in other types of control flow such as loops. However, for ease of discussion, we use only conditions throughout this submission.

The paper is organized as follows; in the next section, we state our model including some assumptions about the detection of errors. In *Section* 3, we compare with related work. We consider the advantage of using a replica as a shadow node in *Section* 4. We discuss compiler support in *Section* 5, followed by analysis of the communication costs in *Section* 6. In the following section, we discuss recovery from various failure scenarios. An extension of our proposed model using four replicas is explained in *Section* 8. Experimental results are presented in *Section* 9. Finally, we conclude with future directions.

2 Model and Assumptions

A distributed real-time application consists of a set of communicating processes or objects allocated to different execution nodes. In a fault-tolerant environment, the term *computing station* is used to refer to an execution node and its associated redundancies (replicas), see for example [18, 19]. We use primary-backup model of fault tolerant architecture with three nodes per computing station (*Figure* 4). Nodes within the computing station are fully connected by communication links. Messages can flow in either direction. We assume a reliable communication protocol with an upper bound on message delivery delay.

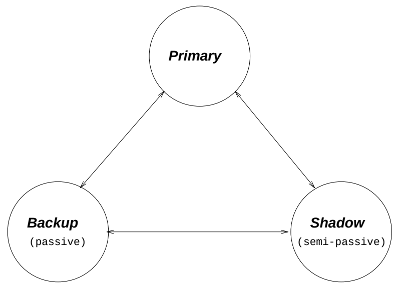

Figure 4: Three Nodes Fault-Tolerant Computing Station

The processor of every node satisfies the fail-stop assumption: that is, it halts upon failure without producing incorrect output [25]. We assume that any error will be detected internally within the computing station using special hardware or through timing out

```
        ORIGINAL                     PRIMARY COPY                      SHADOW COPY

    if (exp)                     Activate backup                   fork block1
      block1                     if (exp)                          if (exp)
    else                           get updates for                   send updates for
      block1                       block1 from shadow                block1 to primary
                                 else                              else
                                   block2                            block2
    /* Original code */      /* Transformed code on            /* Transformed code on the
                                the primary server */             semi-passive backup */
```

Figure 3: Speculative execution for parts of the code on the semi-passive backup.

the delivery of messages, as assumed in [24]. Communication between processes is in the form of local or remote calls.

3 Related Work

Fault-tolerant systems usually fall into two classes, active and passive replication. Active replication has been used in some experimental real-time systems projects, including MARS [19] and Delta-4 XPA [28]. Previous work on passive replica largely entails rollback recovery while minimizing overhead due to checkpointing.

The passive replica model relies on saving the state of each process during failure-free execution. After failure, if a consistent system state can be formed using individual process states previously saved, the fault can be recovered [8]. There are two approaches for rollback recovery. The first uses consistent checkpointing [9, 10, 17], with a coordinator to synchronize writing updates of every process to stable storage. To recover from failure, a global system consistent state reflecting a consistent state for every individual process, is built; some processes may have to be rolled back to an earlier state from their most recent checkpoints. A new consistent checkpoint must be recorded before committing any output. This makes the frequency of checkpointing high and consequently increases the overhead. On the other hand, rollback recovery can be implemented using optimistic message logging as in [15, 27]. Optimistic message logging avoids frequent checkpointing by asynchronously buffering messages received by processes. Recovery replays the messages in the buffer. Processes still need to save state, but not as frequently and without coordination. In our model, we use a variant of consistent checkpointing, that is, we use backups for saving state. Recovery is initiated within the computing station itself. We do not rely on the existence of a coordinator, but use compiler analysis to insert checkpoints.

The performance penalty largely from consistent checkpointing in fault-tolerant distributed systems has been measured in [10]. The experiments reported show that the execution time increases by 1-5.8%, where fast hard drives were used as stable storage. These results are based on long-running distributed applications. Although they did not address real-time issues, this conclusion is very encouraging. We expect to get less of a performance penalty because we are not using permanent storage for checkpointing.

The idea of window-consistent replication service is evaluated in [24]. This minimizes the frequency of updates by relaxing the consistency constraints of the replicated data. Consistency requirements for the data are provided by the application in terms of a time window. In case of failure, it is possible to rely on the most recent data to recover. This approach to minimizing the overhead can be combined with our work; using a time-window during compilation can relax the update frequency.

Speculative execution has been addressed on different levels. The lowest level is the machine instruction level in super-scalar and VLIW machines, as for example [2]. Statement level speculative execution through safe code motion, using data flow analysis techniques, has been used to enhance schedulability of real-time processes in [11, 13]. Forking new processes to speculatively execute blocks of the code without destroying timeliness during rollback is presented in [29]. Compile-time analysis is used to detect safe opportunities for speculative execution and transform the code to fork such new processes. A semi-static approach to speculative execution is presented in [20]. The approach is to use profiling to collect information about the correlation between branch conditions and replicate the code of the most probable branches. In addition, speculative execution has been successfully used to achieve timeliness in real-time databases [3].

4 Semi-passive Replica

The computing station in our model consists of three nodes, *Figure* 4. One node acts as a primary server. During fault-free execution, the primary server reacts to client requests and sends update messages (checkpoints) to other nodes within the same computing station. The frequency of checkpoints will be determined by the compiler, as elaborated in *Section* 5. The other two nodes are the semi-passive shadow and the passive backup.

The passive backup does not execute any code during normal execution; it just receives update messages from the primary server. The backup acts as a storage device keeping a copy of the data from the primary; it also has a copy of the code loaded. All updates from the primary will be used to upgrade backup copy of the data region. One of the attractive features in our model is the use of dynamic memory instead of fixed disks to store checkpoints, significantly reducing the overhead of checkpointing.

The shadow server is used for performing speculative execution as directed by the primary node. It acts as a second passive node when there is no code to be executed speculatively. When the primary server sends an activation message to the shadow, it forks a process sharing the same data region with the original non-running replica of the process on the primary. The new process starts executing a block of code. Upon conclusion, the shadow sends the state update back to the primary node. The primary server may commit the results or simply ignore the message if the speculatively executed code would not in fact have been executed. The state of the shadow will not be up-to-date until the next checkpoint from the primary. In addition, if the primary fails while the shadow is executing, we can not trust the shadow state to perform the recovery. These situations motivate the need for another passive backup in our proposed computing station.

Compiler techniques are used to identify locations in the code when update messages are to be sent, as discussed in the next section.

5 Compiler Support

We rely intensely on compiler support to investigate opportunities for speculative execution as well as minimizing checkpointing overhead. Various compiler optimization techniques can be applied. Moreover, we use compile-time analysis to mark points in the code suitable for sending update messages to other replicas in the computing station. In this section, we address these issues.

During compilation, profitable and safe opportunities for speculative execution are detected. Speculative execution is an optimistic execution of part of the code based on assumptions that have not yet been validated. For example, we can speculatively execute the largest or most probable branch in a conditional before the evaluation of the condition. Moreover, we can speculatively run the next loop iteration while the execution of the current iteration is still in progress. We have addressed the compile-time prediction of safe and profitable speculative execution and developed compiler transformation rules in [29].

Using the shadow node for running some code speculatively affects checkpointing. Before speculatively executing part of the code on the shadow, its state should be verified to be sufficient for starting the execution properly. Various data flow techniques can be used to check whether the data values to be referenced in the speculative code are the most recent. In *Figure* 1, for example, in the primary copy, we need to send an update message to the shadow before activating it to upgrade the data used in *block1* to the most recent version computed on the primary. The speculative transformation rules make sure that the code to be run on the shadow node does not require values being computed by the primary server. For example, if the primary is executing a function, the code that is speculatively executed by the shadow should use neither the return value nor other values modified by that function. If some values need to be updated, the primary must send them before activation of the speculative process on the shadow. Considering the code in *Figure* 5, data flow analysis techniques can be used to detect that x and y have been modified since the last update, and therefore they need to be included in the next checkpointing message. Using interprocedural analysis techniques, we make sure that both x and y are not going to be modified in the function $f()$. The compiler will insert a checkpoint before the conditional by sending an update message to the shadow with the latest values of x and y and send an activation for a process which corresponds to the *then-clause*. The calls to the functions $q()$ and $g()$ are replaced by receiving the result message. Again the compiler will detect which values are modified and include them in the message. If $f(x, y)$ is determined to be false, the compiler will include both j and t in the next update message to the shadow. Because the *else-clause* is not part of the worst-case execution path (since it was not selected for speculative execution), the deadline will not be missed. Then the compiler will evaluate the tradeoffs and transform only if it is not extending the deadline.

```
 ORIGINAL                  PRIMARY COPY

x = x + 1;                x = x + 1;
y = y * z;                y = y * z;
if (f(x,y))               send_shadow(x,y);
  j = q(x,y,z);           activate_shadow(p1);
  t = g(y,r,n);           if (f(x,y))
else                        receive_shadow(j,t);
  j = x * y * z;          else
                            j = x * y * z;
                            send_shadow(x,y);
```

Figure 5: Compiler inserted checkpointing to support speculative execution.

An important issue is to avoid blocking the primary waiting for a shadow to commit. This may occur when performing more time consuming computation on the shadow than that required by primary to conclude the condition. If the shadow computation fails, the primary will spend time waiting for the results, which will extend the worst-case path. To avoid such an unsafe scenario, compile-time checks need to be performed. A shadow should be activated early enough so that it will finish before the primary finishes evaluating the condition. If this is not possible, due to data dependence, the primary should start executing the branch up to a time equivalent to the delay expected for the results to be available. If the shadow fails, the primary can safely continue.

Additional checkpoints may be required for consistency. Checkpointing is performed before committing output to the outside world (external to the computing station), as in [10]. However, this is not effective for real-time systems, as recovery from failure may need a long rollback, which may destroy the timing requirements. Another possible alternative is to use a time-window to determine the frequency of the updates. That window can be drawn from the consistency semantics of the system, in the sense of [24]. However, this approach schedules a periodic process to perform checkpointing without using incremental update. To keep the data window-consistent, we consult a tool for static-time prediction of the execution time of the system to insert checkpoints in the code during compilation.

Further optimization techniques can be used to decrease the overhead and enhance the schedulability of updates. Incremental update of the internal state can be used instead of sending the whole state. The compiler detects the modified values and includes them only in the next update message. Hardware support can accelerate this process through dirty bits and copy-on-write mechanisms [22]. It is shown in [10] that incremental checkpointing can decrease the overhead substantially for a wide range of applications. In addition, safe application of code motion compiler techniques, as discussed in [11, 13], can be used to insert checkpoints safely.

Moreover, the compiler can detect busy-idle profiles, as discussed in [14], to interleave the sending of updates to both replicas. For example, if the primary is making a remote call to another computing station, it can update the replicas while waiting for the results. In addition, the primary can also interleave message sending to the passive backup with the speculative execution of a long block by the shadow. In the example in *Figure* 1, assume that *block1* executed by the shadow takes longer time than evaluating the condition by the primary. If the primary is going to commit a result which is not yet available from the shadow, the primary can meanwhile update the passive backup. The compiler consults the execution time prediction tool as well as the timing constraint to avoid extending any deadline.

Thus, using compiler inserted checkpoints in combination with compiler optimization techniques can be used to minimize overhead. In addition, opportunities for profitable speculative execution can be extracted, enhancing performance and absorbing the update overhead.

6 Communication Costs

We assume a direct communication link between all nodes within the computing station. There are two kinds of messages: data and control messages. Data (checkpointing) messages flow from the primary server to the shadow and backup. In addition, the shadow can send the results of the speculative execution back to the primary. Control messages are used to activate the shadow to execute part of the code. The compiler generates processes for every block of code subject to speculative execution. The activation message will refer to the appropriate process as opposed to the primary. In the code in *Figure* 5, the primary sends an incremental update to the shadow (values of x and y), followed by a control message. The activation message refers to a process which reflects the code in the *then-clause*. The shadow sends the state changes back to the primary upon completion of the speculative execution.

In case of failure of the primary, either the shadow or the backup will take over according to the failure scenario as discussed in the next section. If the shadow takes over after primary failure, update messages still flow to the backup; similarly, if the backup is used, messages will be sent to the shadow. The backup does not send messages to the primary except for acknowledging receipt of the checkpoint, if necessary, to guarantee reliable delivery of update messages.

Having direct and dedicated communication links makes transmission of messages fast and avoids contention and routing delays. In addition, special hardware (communication coprocessor) can be used. Our basic assumption is that the communication time is predictable (which is required anyway for schedulability analysis) and reasonably fast to enable additional feasible speculative execution opportunities.

7 Recovery from Failure

As stated in *Section* 2, we assume faults can be detected internally within the computing station. During execution, we can anticipate the following failure scenarios: primary fails while the shadow is not executing, primary fails while the shadow is speculatively executing a block of code, shadow fails while executing, shadow fails while being passive, and backup fails. We address recovery mechanisms for each scenario according to the location of the fault.

Recovery from Primary Failure: If the primary node fails, service should be resumed on one of the replicas. Recovery from fault will be initiated according to the state of the shadow node. If the shadow node was idle before failure, execution can be resumed on either node, shadow or passive backup. However, the shadow node may be preferred, as failure may have occurred just after committing the results of speculative execution. Thus the shadow will be in a forward state in terms of execution progress, and there will be additional time to adapt the recovery procedure without missing deadline. Regardless, until the primary recovers, the original code should be used, rather than the transformed speculative code, to provide continued fault tolerance. On the other hand, if the shadow node was speculatively executing some code, service can not resume on it. Here the execution state of the shadow will be unreliable, as it is based on assumptions which have not been validated before the failure. Considering example in *Figure* 1; if the primary fails while evaluating the *exp*, there is no way to predict whether to commit or reject the results of the shadow execution. In this scenario, the passive backup node should be used to recover. However, the passive backup can still benefit from the shadow execution performed. As we discussed in *Section* 5, state updates will be sent before executing the if-statement, so even in that worst case of recovery the backup does not have to roll back from an old state.

Recovery from Shadow Failure: The primary node is the main provider of service. If the shadow node fails, service can still be offered, but without speculative execution. The original copy of the code should be used instead of the transformed copy on the primary node. In this situation, the primary will keep updating only the passive backup. An interesting scenario is when the shadow node fails while executing some speculative code. As we discussed in 5, the compiler will try to start the shadow early enough to finish before the conclusion of the condition. On the other hand, if it is not possible to start the shadow early, the primary will execute the branch without being blocked for the results. Thus, the primary will continue without delay executing the code and continue in a serial manner. Because the primary was busy during the speculative execution before the shadow fails, there will be minimal performance degradation if any.

Recovery from Backup Failure: Similar to shadow failure: if the passive backup node fails, the primary node can still provide service. However, all speculative execution should be disabled and the shadow then acts as a passive backup.

8 Four Processor extension

In this section we discuss the benefit of computing station of four nodes, *Figure* 6. As for the three-node computing station, nodes are directly connected through bidirectional links. The extra node is used as an additional shadow server. We show not only that reliability increases due to the higher redundancy, but also that performance can be enhanced for both average-case and worst-case execution.

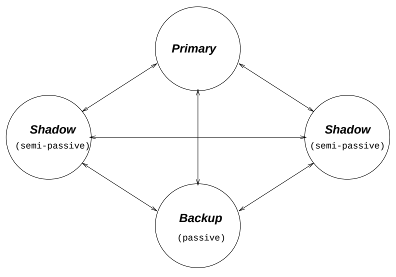

Figure 6: Four Nodes Fault-Tolerant Computing Station

With additional shadow nodes, it is possible to speculatively execute both branches of a conditional; the *then-clause* can be executed on one shadow and the *else-clause* on the other shadow. The primary will commit the results from one of the shadows. Using this mechanism, worst-case performance will be enhanced due to parallel execution of the branches. With while loops, we can execute three iterations in parallel (given that no data dependence). Moreover, the shadow which has the committed results can be used to propagate updates to the other two replicas, releasing the primary from that overhead.

In addition, the extra node can make recovery from failure more efficient. If the primary fails, the three other nodes can still be used as a three-node computing station. Moreover, most of the time we will be performing forward recovery, since when the primary fails while the shadow is executing some code speculatively, the backup can take over and re-evaluate the condition of the if-statement and still commit the results from one of the shadows. This error scenario required backward recovery in case of three-node station, as shown in *Section* 7.

9 Experimental Results

In this section, we examine the performance of speculative execution using the three-node computing station. The experiment consists of four stages. First, we generate input programs using a workload generator. Each program consists of statements selected out of seven types, *assignment, if, read, write, while, blocking call and non-blocking call.* Two sets of variables are associated with each statement: referenced variables, and modified variables. Second, execution time is attached to each statement in a program. Third, we run the programs and classify them into 4 groups.

Group 0: programs which miss deadlines with and without speculative execution.

Group 1: programs which meet deadlines with and without speculative execution.

Group 2: programs which meet deadlines using speculative execution but miss deadlines without it.

Group 3: programs which miss deadlines using speculative execution but meet deadlines without, or in which no speculative execution can be performed.

In the fourth step, programs of group 1 and group 2 are run and faults are randomly injected into programs to see how much speculative execution can help in reducing the percentage of missing deadlines. The programs of group 0 are of no interest; if the program cannot meet its deadline even with speculative execution, there may be a problem in the design. No programs belong to group 3 since no code will be speculatively executed in the shadow if it is not safe or profitable according to the analysis performed by the compiler.

The workload generator generates 1000 programs and each runs for 10 times. The worst-case execution time (WCET) of a program is determined from static timing estimate on its flow graph. The selection of deadlines affects the size of each group. For instance, there are 55%, 0%, 45% and 0% of groups 0, 1, 2, 3, respectively, where the deadline is in the range of

$[.6, .8] *$ WCET. However, there are 4%, 38%, 58% and 0% of groups 0, 1, 2, 3, respectively, provided that the deadline is in the range of $[.8, 1.1] *$ WCET. The reported results in *Figure* 7 are based on selecting the deadline randomly in the range of $[0.75, 1.0] *$ WCET. So among the 1000 programs there are 3% of group 0, 37% of group 1 and 60% of group 2.

Based on our experience with real-time programs [26], we decided to use a frequency of 6% to 10% of *if* statements per generated program. Although a fault may occur in primary, shadow or backup, we assume that faults will not propagate from one node to another. Little penalty is anticipated if a fault occurs either on the backup node, or on the shadow node while no speculative execution is running. If a program meets the deadline both with and without speculative execution, it will probably meet that deadline after injecting a single fault, since, in our current model, the time cost of a single fault is not significant. We therefore decided not to report group 1, although we still see small gains with the use of speculative execution. *Figure* 7 summarizes the results of group 2 injecting a single fault. We anticipate that much more significant gains would result with the use of speculative execution either by allowing a series of isolated faults, or by increasing the possible cost of a single fault. The following are clarification of the notation used in the table in *Figure* 7.

1. *Pgm Size:* the number of statements per program.
2. *Avg_if_blk Size:* the average size of an if-block.
3. *# missing w/SE:* using speculative execution, the number of programs missing deadline over the total number of group 2. For example, on the second row the number 5019/5739 denotes that among 5739 of group 2 there are 5019 cases missing deadlines after injecting faults.
4. *# missing w/o SE:* with no speculative execution, the number of programs missing deadlines over the total number of group 2. For instance, on the second row the number 5739/5739 denotes that without speculative execution all the programs (5739) of group 2 miss deadlines after injections of faults.
5. *Speedup:* the percentage of $1 - \frac{T_{SE}}{T_{NSE}}$ where T_{SE} and T_{NSE} denote the execution time of a program with and without speculative execution respectively.
6. *Imprv:* is the reduction in the percentage of missed deadlines by speculative execution.

¿From the above, we conclude that on the average speculative execution of if-blocks reduces about 19% of missing deadlines and speeds up the execution of programs by 27%. The sizes of if-blocks determine the opportunities of speculative execution. In this implementation the opportunities are about 90% of the 6% to 10% of if statements. The shadow utilization is about 30% which indicates that a shadow node is under-utilized and can serve multiple primary nodes. We have made other runs with the same number of statements and different if-block sizes, the results in average are not significantly different. For a detailed discussion about various factors affecting the applicability of speculative execution to real-time programs, the reader can refer to [30].

While speculative execution trades resources for average-case performance, we have shown that, at least in a limited model, it can also improve both WCET and likelihood of deadline satisfaction in the presence of faults. Although we cannot in general guarantee timeliness, we can reduce both the incidence and cost.

10 Conclusion and Future Work

We have shown that compiler assisted speculative execution can be used to achieve fault-tolerance in real-time systems. We proposed a computing station based on passive and semi-passive replicas. We argued that using speculative execution can enhance the average-case performance and sometimes even the worst-case performance. That enhancement in performance can be used to absorb the overhead due to checkpointing. Moreover, we minimized the time required for checkpointing by using the replica's main memory instead of fixed disk storage. We showed that we can achieve forward recovery most of the time (all the time if we have four replica).

In this paper, we assume that faults are unrelated and independent. In the future, we plan to handle linked failure and multiple faults. We also intend to apply the analysis of [23] to implement recovery as exception handling. In addition, we will try to apply some dependability evaluation methods, such as [1], to formally evaluate our ideas. Combining these two analyses, we expect to be able to guarantee an upper bound on the likelihood of a catastrophic process fault, that is, one in which the system keeps running, but deadlines are missed. Finally, we will investigate the possibility of using the same shadow node for multiple primary processors.

References

[1] J. Arlat, et al., "Fault Injection and Dependability Evaluation of Fault-Tolerant Systems," *IEEE Transactions on Computers,* Vol. 42, No. 8, pp. 913–923, August 1993.

[2] N. Alewine, W. Fuchs and W. Hwu, "Application of compiler-assisted rollback recovery to speculative execution repair," *Proceedings of the Conf. on Hardware and Software Architectures for Fault Tolerance Experiences and Perspectives,* Le Mont Saint Michel, France, 1993.

[3] A. Bestavros and S. Braoudakis, "Timeliness via Speculation for Real-Time Databases," *Proceedings of* 15^{th} *IEEE Real-Time Systems Symposium,* San Juan PR, 1994.

[4] K. Birman, "The process group approach to reliable distributed computing," *Communications of the ACM,* Vol. 36, No. 12, pp. 37–53, December 1993.

Pgm Size	Avg_if_blk	# missing w/ SE	# missing w/o SE	Speedup	Imprv
1500	21	5019/5739	5739/5739	17.14%	12.5%
1500	32	5505/6556	6556/6556	23.07%	16.0%
2000	32	5451/6543	6543/6543	23.16%	16.7%
2000	58	3698/5737	5737/5737	31.48%	35.4%
5000	32	5821/6653	6653/6653	22.98%	12.5%
5000	58	4012/5279	5279/5279	34.40%	24.0%
8000	60	3604/4822	4822/4822	37.84%	25.3%

Figure 7: Summary of Experimentation Results

[5] A. Broggi, "A Novel Approach to Lossy Real-Time Image Compression: Hierarchical Data Reorganization on a Low-cost Massively Parallel System," *Journal of Real-Time Imaging,* Academic Press, (to appear).

[6] K. Birman, T. Joseph, T. Raeuchle, and A. Abbadi, "Implementing Fault-Tolerant Distributed Objects," *IEEE Trans. on Software Engineering,* Vol. SE-11, No. 6, 1985.

[7] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg, "Primary-backup protocols: Lower bounds and optimal implementations," *Proceedings of IFIP Working Conference on Dependable Computing,* pp. 187–198, 1992.

[8] K. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed systems," *ACM Trans. on Computer Systems,* Vol. 3 no. 1, pp. 63–75, 1985.

[9] F. Cristian and F. Jahanian, "A timestamp-based checkpointing protocol for long-lived distributed computations," *Proceedings of the* 10th *Symposium on Reliable Distributed Systems,* pp. 12–20, September 1991.

[10] E. Elnozahy, D. Johnson and W. Zwaepepoel, "The Performance of Consistent Checkpointing," *Proc. of the* 11th *IEEE Symposium on Reliable Distributed Systems,* 1992.

[11] R. Gerber and S. Hong, "Compiling Real-Time Programs with Timing Constraints Refinement and Structural Code Motion," *IEEE Transactions of Software Engineering,* Vol. 21, No. 5, May 1995.

[12] C. Giardina and E. Dougherty, *Morphological Methods in Image and Signal Processing.* Prentice Hall, Englewood Cliffes NJ 1988.

[13] P. Gopinath and R. Gupta, "Applying Compiler Techniques to Scheduling in Real time Systems," *Proceedings of the* 11th *IEEE Real-Time Systems Symposium,* pp. 247–256, Orlando, Florida, December 1990.

[14] R. Gupta and M. Spezialetti, "Busy-Idle Profiles and Compact Task Graphs: Compile-time Support for Interleaved and Overlapped Scheduling of Real-Time Tasks," *Proc. of the* 15th *IEEE Real-Time Systems Symposium,* 1994.

[15] D. Johnson, "Efficient Transparent Optimistic Rollback Recovery for Distributed Application Programs," *Proceedings of the* 12th *IEEE Symposium on Reliable Distributed Systems,* October 1993.

[16] K. Kim, "Design of Real-Time Fault-Tolerant Computing Stations," *Proceedings of the NATO Advanced Study Institute of Real Time Computing,* Sint Maarten, Dutch Antilles, October 1992.

[17] J. Kim and T. Park, "An Efficient Protocol for Checkpointing Recovery in Distributed Systems," *IEEE Transactions on Parallel and Distributed Systems,* Vol. 4, No. 8, pp. 955-960, August 1993.

[18] K. Kim and H. Welch, "Distributed Execution of Recovery Blocks: An Approach for Uniform Treatment of Hardware and Software Faults in Real-Time Applications," *IEEE Transactions on Computers,* pp. 626–636, May 1989.

[19] H. Kopetz, et al., "Distributed Fault-tolerance Real-Time Systems: The Mars Approach," *IEEE Micro,* pp. 25-39, February 1989.

[20] A. Kral, "Improving Semi-static Branch Prediction by Code Replication," *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation,* ACMPRESS, pp. 97–105, June 1994.

[21] D. Leinbaugh, and M. Yamini, "Guaranteed Response Times in a Distributed Hard Real Time Environment," *IEEE Transactions on Software Engineering,* Vol. SE-12, No. 12, pp. 1139–1144, December 1986.

[22] K. Li, J. Naughton, and J. Plank, "Real-time concurrent checkpoint for parallel programs," *Proceeding of the 1990 Conference on the Principles and Practice of parallel Programming,* pp. 79–88, March 1990.

[23] T. J. Marlowe, A. D. Stoyenko, S. P. Masticola, L. R. Welch, "Schedulability-Analyzable Exception Handling for Fault-Tolerant Real-Time Languages," *Journal of Real-Time Systems,* Vol. 7, pp. 183–212, 1994.

[24] A. Mehra, J. Rexford, H. Ang, and F. Jahanian, "Design and Evaluation of a Window-Consistent Replication Service," *Proceedings of the first IEEE Real-Time Technology and Applications Symposium,* May 1995.

[25] R. Schlichting, and F. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems,* Vol. 1, No. 3, pp. 222–238, August 1983.

[26] A. D. Stoyenko, T. J. Marlowe, W. A. Halang, M. Younis, "Enabling Efficient Schedulability Analysis through Conditional Linking and Program Transformations," *Control Engineering Practice,* Vol. 1, No. 1, pp. 85–105, Jan 1993.

[27] R. E. Strom, S. A. Yemini. "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems,* Vol. 3, No. 3, pp. 204–226, August 1985.

[28] P. Verissimo, et. al., *The extra performance architecture (xpa),* In D. Powell, editor, Delta-4 - A generic Architecture for Dependable Distributed Computing, 1991.

[29] M. Younis, T. Marlowe, and A. Stoyenko, "Compiler Transformations for Speculative Execution in a Real-Time System," *Proceedings of the* 15th *Real-Time Systems Symposium,* San Juan, Puerto Rico, December 1994.

[30] M. Younis, G. Tsai, T. Marlowe, and A. Stoyenko, "Statically Safe Speculative Execution For Real-Time Systems," *Technical Report CIS-95-30,* Computer and Information Science Dept., New Jersey Institute of Technology, 1995.