Flexible Rollback Recovery in Dynamic Heterogeneous Grid Computing

Samir Jafar,[*] Axel W. Krings [†] and Thierry Gautier [‡]

[*]Damascus University, Syria
[†]University of Idaho, USA
[‡]ID-IMAG, France

Abstract

Large applications executing on Grid or cluster architectures consisting of hundreds or thousands of computational nodes create problems with respect to reliability. The source of the problems are node failures and the need for dynamic configuration over extensive run-time. This paper presents two fault-tolerance mechanisms called Theft Induced Checkpointing *and* Systematic Event Logging. *These are transparent protocols capable of overcoming problems associated with both, benign faults, i.e., crash faults, and node or subnet volatility. Specifically, the protocols base the state of the execution on a dataflow graph, allowing for efficient recovery in dynamic heterogeneous systems as well as multi-threaded applications. By allowing recovery even under different numbers of processors, the approaches are especially suitable for applications with need for adaptive or reactionary configuration control. The low-cost protocols offer the capability of controlling or bounding the overhead. A formal cost model is presented, followed by an experimental evaluation. It is shown that the overhead of the protocol is very small and the maximum work lost by a crashed process is small and bounded.*

1. Introduction and Motivation

Grid and cluster architectures have gained popularity for computationally intensive parallel applications. However, the complexity of the infrastructure, consisting of computational nodes, mass storage and interconnection networks, poses great challenges with respect to overall system reliability. Simple tools of reliability analysis show that as the complexity of the system increases, its reliability, and thus Mean Time to Failure (MTTF), decreases. If one models the system as a series reliability block diagram [30], the reliability of the entire system is computed as the product of the reliabilities of all system components. For applications executing on large clusters or a Grid, e.g., Grid5000 [13], the long execution times may exceed the MTTF of the infrastructure and thus render the execution infeasible. As an example let us consider an execution lasting 10 days in a system that does not consider fault-tolerance. Under the optimistic assumption that the MTTF of a single node is 2000 days, the probability of failure of this long execution using 100, 200, or 500 nodes is 0.39, 0.63 or 0.91 respectively, approaching fast certain failure. The high failure probabilities are due to the fact that, in the absence of fault-tolerance mechanisms, the failure of a single node will cause the entire execution to fail. Note that this simple example does not even consider network failures, which are typically more likely than computer failure. Fault-tolerance is thus a necessity to avoid failure in large applications, such as found in scientific

computing, executing on a Grid or large cluster.

The fault-tolerance mechanisms also have to be capable of dealing with the specific characteristics of a heterogeneous and dynamic environment. Even if individual clusters are homogeneous, heterogeneity in a Grid is mostly unavoidable, since different participating clusters often use diverse hardware or software architectures [13]. One possible solution to address heterogeneity is to use platform-independent abstractions such as the Java Virtual Machine. However, this does not solve the problem in general. There is a large base of existing applications that have been developed in other languages. Re-engineering may not be feasible due to performance or cost reasons. Environments like Microsoft .Net address portability but only few scientific applications on Grids or clusters exist. Whereas Grids and clusters are dominated by unix operating systems, e.g. Linux or Solaris, Microsoft .Net is Windows-centric with only recent or partial unix support.

Besides heterogeneity one has to address the dynamic nature of the Grid. Volatility is not only an intra-cluster issue, i.e., configuration changes within a cluster, but also an inter-cluster reality. Intra-cluster volatility may be the result of node failures, whereas inter-cluster volatility is caused by network disruptions between clusters. From an administrative viewpoint the reality of Grid operation, such as cluster/node reservations or maintenance, may restrict long executions on fixed topologies due to the fact that operation at different sites may be hard to coordinat. It is usually difficult to reserve a large cluster for long executions, let alone scheduling extensive *uninterrupted* time on multiple, perhaps geographically dispersed, sites. Lastly, configuration changes may be induced by the application as the result of changes of run-time observable Quality of Service (QoS) parameters.

To overcome the aforementioned problems and challenges, we present mechanisms that tolerate faults and operation-induced disruption of parts or the entire execution of the application. We introduce flexible rollback recovery mechanisms that impose no artificial restrictions on the execution. They do not depend on the pre-failure configuration and consider (1) node and cluster failures as well as operation-induced unavailability of resources and (2) dynamic topology reconfiguration in heterogeneous systems.

The reminder of the paper is organized as follows: In Section 2 we present the necessary background

information and related work. Next, in Section 3 we describe the execution model considered. Two rollback-recovery protocols are introduced in Section 4 and Section 5. A theoretical performance and cost analysis of these protocols is presented in Section 6, followed by an experimental validation of the theoretical results in Section 7. Finally, we conclude the paper in Section 8.

2. Background

Several fault-tolerance mechanisms exist to overcome the problems described in Section 1. Each fault in a system, may it be centralized or largely distributed, has the potential for loss of information, which then has to be re-established. Recovery is thus based on redundancy. Several redundancy principles exit, i.e., time, spatial and information redundancy. Time redundancy relies on multiple executions skewed in time on the same node. Spatial redundancy, on the other hand, uses physically redundant nodes for the same computations. The final result is derived by voting on the results of the redundant computations. However, there are two disadvantages associated with redundancy:

1. Only a fixed number of faults can be tolerated depending on the type of fault. This number of redundant computations depends on the fault model, which defines the degree of replication needed to tolerate the faults assumed [18, 29]. The exact types of faults considered, e.g. crash fault or omission fault, and their behavior will be described later in Subsection 3.4.

2. The necessary degree of redundancy may introduce unacceptable cost associated with the redundant parallel computations and its impact on the infrastructure [24]. This is especially true for intensive Grid computations [2].

As a result, solutions based on replication, i.e., time and spatial redundancy, are, in general, not suitable for Grid computing where resources are preferably used for the application itself.

In information redundancy, on the other hand, redundant information is added that can be used during recovery to reconstruct the original data or computation. This method is based on the existence of the concept of stable storage [10]. One has to note that stable storage is only an abstraction whose implementation depends on the fault model assumed. Implementations of stable storage range from

simple local disks, e.g., to deal with the loss of information due to transient faults, to complicated hybrid-redundancy management schemes, e.g., configurations based on RAID technology [21] or survivable storage [32].

We consider two methods based on stable storage, i.e., logging and checkpointing.

2.1 Logging-based Approaches

Logging [1] can be classified as pessimistic, optimistic or causal. It is based on the fact that the execution of a process can be modeled as a sequence of state intervals. The execution during a state interval is deterministic. However, each state interval is initiated by a nondeterministic event [27]. Now assume that the system can capture and log sufficient information about the nondeterministic events that initiated the state interval. This is called the *piecewise deterministic* assumption [27] (PWD). Then a crashed process can be recovered by (1) restoring it to the initial state and (2) replaying the logged events to it in the same order they appeared in the execution before the crash. To avoid a rollback to the initial state of a process and to limit the amount of nondeterministic events that need to be replayed, each process periodically saves its local state. Log-based mechanisms in which the only nondeterministic events in a system are the reception of messages is usually referred to as *message logging*.

Examples of systems based on message logging include MPICH-V2 [7], and FTL-Charm++ [8]. A disadvantage of log-based protocols for applications with extensive inter-process communication is the potential for large overhead with respect to space and time, due to the logging of messages.

2.2 Checkpointing-based Approaches

Rather than logging events, checkpointing relies on periodically saving the state of the computation to stable storage [9]. If a fault occurs, the computation is restarted from one of the previously saved states. Since the computation is distributed, one has to consider the tradeoff space of local and global checkpointing strategies and their resulting recovery cost. Thus, checkpointing-based methods differ in the way processes are coordinated and in the derivation of a consistent global state. The consistent

global state can be achieved either at the time of checkpointing or at the time of rollback recovery. The two approaches are called coordinated and uncoordinated checkpointing respectively.

Coordinated checkpointing requires that all processes coordinate the construction of a consistent global state before they write the individual checkpoints to stable storage. The disadvantage is the large latency and overhead associated with coordination. Its advantage is the simplified recovery without rollback propagation and minimal storage overhead, since each process only needs to keep the last checkpoint of the global "recovery line". This kind of protocol is used e.g, in [26, 33].

Uncoordinated checkpointing on the other hand assumes that each process independently saves its state and a consistent global state is achieved in the recovery phase [10]. The advantage of this method is that each process can make a checkpoint when its state is small. However, there are two main disadvantages. First, there is a possibility of rollback propagation which can result in the domino effect [23], i.e., a cascading rollback to the beginning of the computation. Second, due to the cascading effect the storage requirement is much higher, i.e., each process needs to store multiple checkpoints.

A compromise between coordinated and uncoordinated checkpointing is *communication-induced checkpointing*. To avoid the domino effect that can result from independent checkpoints of different processes, a consistent global state is achieved by forcing each process to take additional checkpoints based on some information piggybacked on the application messages [3]. There are two main disadvantages with this approach. First it requires global rollback. Second, it can result in the creation, and thus storage, of a large number of unused checkpoints, i.e., checkpoints that will never be used in the construction of a consistent global state. An example of a system using this approach is ProActive [4].

The essential issue in checkpointing and logging methods is to determine what information should be stored in the checkpoint or log. This information will determine the properties and suitable environment of the rollback, e.g., homogeneous versus heterogeneous system architecture or static versus dynamic system configuration. A popular checkpointing library used in systems like CoCheck [26], MPICH-V2 [7] and MPICH-CL [7] is the Condor checkpoint library [19]. In Condor the information constituting the checkpoint is the execution state of the process and thus depends on the specific architecture of the

platform which executes the process. As a consequence, rollback is feasible only on an identical platform and it requires the creation of a replacement process. We will present below an approach that overcomes both of these limitations, using an abstract state of the execution represented by a dataflow graph. This generalizes the approach used in the Satin parallel programming environment [31], which will be further discussed in Subsection 7.5.

3. Execution Model

The general execution model of large Grid applications can be viewed as having two levels, as shown in Figure 1. Level 0 only creates the "abstraction of the execution state" of the application. This abstraction is then used by Level 1 to actually schedule and thus execute the workload.

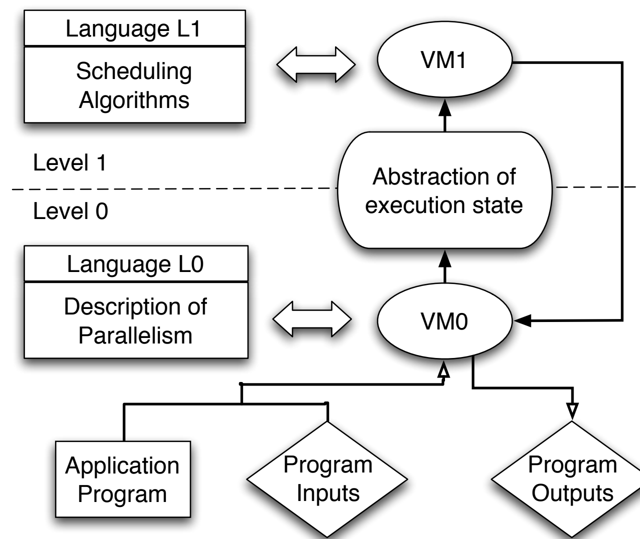

Figure 1. Execution Model

In Level 0 the program to be executed is viewed as an abstraction that represents the state symbolizing the future of an execution. By "future" we mean the execution that has not unfolded yet. Specifically, the input to the virtual machine VM0 is the sequential input program supplemented by instructions for the run-time system that describe the parallelism of the application. This is accomplished by two primitives called *Task_Creation* and *Data_Creation*. Whereas the first creates (but does not execute) an

executable task, the latter creates a shared data object. The sequential program language, together with these primitives, constitutes language L0. Note that L0 is now a language supporting parallelism.

Level 1 takes the abstraction of Level 0 and schedules tasks using the primitives *Task_Export*, *Task_Import* and *Task_Execution*. The sequential program language, together with these primitives, constitutes the language L1. This language encompasses the scheduling algorithm. Consequently, Level 1 implements the dispatcher, whose decisions (which will affect the future of the execution) will be executed at Level 0. In the figure this is indicated with the arrow from the virtual machine VM1 to VM0. Note that both levels represent the run-time system, however, whereas the state of the execution is derived at Level 0, the decisions about the future are made at Level 1.

The justification of the general execution model in Figure 1 is that it is independent of the operating system and the hardware architecture. Furthermore, it does not depend on the number of resources, e.g., processors. As such, the execution model is suitable for heterogeneous and dynamic target systems, e.g., large clusters, Grid or peer-to-peer systems. We will now explain the aforementioned abstraction of the execution state.

3.1. Dataflow representation

The representation of the state of an execution is based on the principle of dataflow [25]. Dataflow allows for a natural representation of a parallel execution and can be exploited for fault-tolerance [20]. In a dataflow model, tasks, which are the smallest units of execution, become ready for execution upon availability of all their input data. The dependencies among tasks are modeled in a dataflow graph, which is defined as a directed graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is a finite set of vertices and $\mathcal{E}$ is a set of edges representing precedence relations between vertices. A vertex $v_i \in \mathcal{V}$ is either a computational task or a shared data object. An edge $e_{ij} \in \mathcal{E}$ represents the dependencies between v_i and v_j. Within the context of this research G is a dynamic graph, i.e., it changes during runtime as the result of task creations/terminations as well as shared data object creations/deletions.

The dynamic dataflow graph should not be confused with the static precedence graphs often used in

scheduling theory. Here, as tasks, data objects and their dependencies are created/deleted, the graph changes. Within the context of the general execution model, graph G is the representation of the global system state, i.e., the "abstraction of the execution state" shown in Figure 1.

Whereas graph G is viewed as a single virtual dataflow graph, its implementation is in fact distributed. Specifically, each process P_i contains and executes a subgraph G_i of G. Thus the state of the entire application is defined by $G = \bigcup G_i$ over all processes P_i. Note that this also includes the information associated with dependencies between G_i and G_j, $i \neq j$. This is due to the fact that G_i, by the definition of the principle of dataflow, contains all information necessary to identify exactly which data is missing.

3.2. Work-stealing

The run-time environment and primary mechanism for load distribution is based on a scheduling algorithm called work-stealing [11, 12]. The principle is simple: when a process becomes idle it tries to *steal* work from another process called *victim*. The initiating process is called *thief*.

Work-stealing is the only mechanism for distributing the workload constituting the application, i.e., an idle process seeks to steal work from another process. From a practical point of view the application starts with the process executing *main()*, which creates tasks. Typically some of these tasks are then stolen by idle processes, which are either local or on other processors. Thus the principal mechanism for dispatching tasks in the distributed environment is task-stealing. The communication due to the theft is the only communication between processes. Realizing that task theft creates the only dependencies between processes is crucial to understand the checkpointing protocol to be introduced later.

With respect to Figure 1, work-stealing will be the scheduling algorithm of preference at Level 1.

3.3. The KAAPI environment

The target environment for multithreaded computations with dataflow synchronization between threads is the *Kernel for Adaptive, Asynchronous Parallel Interface* (KAAPI), implemented as a C++ library. The library is able to schedule programs at fine or medium granularity in a distributed environment.

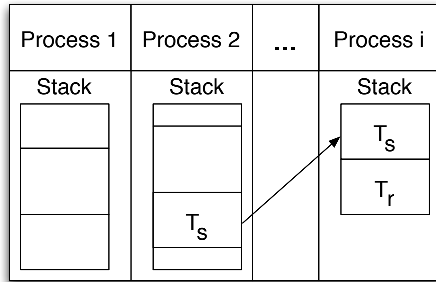

Figure 2. KAAPI **processor model.**

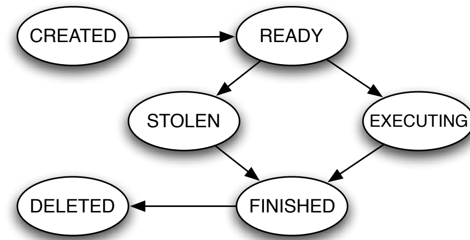

Figure 3. Life-cycle of a task in KAAPI**.**

Figure 2 shows the general relationship between processors and processes in KAAPI. A processor contains one or more processes. Each process maintains its own stack.

The life-cycle of a task in the KAAPI execution model is depicted in Figure 3 and will be described first from a local process' and then from a thief's point of view in the context of task stealing.

At task creation the task enters state *created*. At this time it is pushed onto the stack. When all input data is available the task enters state *ready*. A ready-task which is on the top of the stack can be executed, i.e., it can be popped off the stack, thereby entering state *executing*. A task in the *ready* state can also be stolen, in which case it enters the *stolen* state on the local process, which now becomes a victim. When the task is finished, either on the local process or a thief, it enters state *finished* and proceeds to state *deleted*.

If a task has been stolen, the newly created thief process utilizes the same model. In Figure 2, the theft of task T_s on Process 2 by Process i is shown, as indicated by the arrow. Whereas this example shows task stealing on the same processor, the concept applies also to stealing across processors. On the victim the stolen task is in state *stolen*. Upon theft, the stolen task enters state *created* on the thief. At this instant of time, the stolen task T_s and a task T_r charged with returning the result are the only tasks in the thief's stack, as shown in the figure. Since a stolen task by the definition of work-stealing is ready, it immediately enters state *ready*. It is popped from the stack, thereby entering state *executing*, and upon finishing, it enters state *finished*. It should be noted that the task enters this state on the thief *and* the victim. For the latter this is after receiving a corresponding message from the thief. On both processes

the task proceeds to state *deleted*.

3.4 Fault Model

We will now describe the fault model that the execution model is subjected to. The hybrid fault model described in [29], which defines *benign*, *symmetric* and *asymmetric* faults, will serve as a basis. Whereas benign faults are globally diagnosable and thus self-evident, symmetric and asymmetric faults represent malicious faults which are either consistent or possibly non-consistent. In general, any fault that can be detected with certainty can be dealt with by our mechanisms. On one side this includes any benign fault, such as a crash fault. On the other hand, this considers node volatility [5], e.g., transient and intermittent faults of nodes. It should be noted that results of computation of volatile nodes, which re-join the system, will be ignored.

In order to deal with symmetric or asymmetric faults it is necessary that detection mechanisms are available. Such approaches have been shown in [17, 16] and can be theoretically incorporated in this work.

4 Theft Induced Checkpointing

As seen in the previous section, the dataflow graph constitutes a global state of the system. In order to use its abstraction for recovery, it is necessary that this global state also represents a *consistent* global state.

With respect to Figure 1, we can capture the abstraction of the execution state at two extremes. At Level 0 one assumes the representation derived from the construction of the dataflow graph, whereas at Level 1 the interpretation is derived as the result of its evaluation, which occurs at the time of scheduling.

In this section we will introduce a Level 1 protocol capable of deriving a fault-tolerant coherent system state from the interpretation of the execution state. Specifically, we will define a checkpointing protocol called *Theft Induced Checkpointing*, (TIC).

4.1. Definition of a checkpoint

As indicated before, a copy of the dataflow graph G represents a global checkpoint of the application. In this research, checkpoints are with respect to a process, and consist of a copy of its local G_i, representing the process' stack. The checkpointing protocol must ensure that checkpoints are created in such a way that G is always a consistent global application state, even if only a single process is rolled back. The latter indicates the powerful feature of individual rollbacks.

The checkpoint of G_i itself consists of the entries of the process' state, e.g., its stack. As such, it constitutes its tasks and their associated inputs, and not the task execution state on the processor itself. Understanding this difference between the two concepts is crucial. Checkpointing the tasks and their inputs simply requires to store the tasks and their input data as a dataflow graph. On the other hand, checkpointing the execution of a task usually consists of storing the execution state of the processor as defined by the processor context, i.e., the processor registers such as program counters and stack pointers as well as data. In the first case, it is possible to move a task and its inputs, assuming that both are represented in a platform-independent fashion. In the latter case the fact that the process context is platform-dependent requires a homogeneous system in order to perform a restore operation or a virtualization of this state [28].

The jth checkpoint of process P_i will be denoted by CP_i^j. Thus the subscript denotes the process and the superscript the instance of the checkpoint.

4.2. Checkpoint protocol definition

The creation of checkpoints can be initiated by (1) work-stealing or (2) at specific checkpointing periods. We will first describe the protocol with respect to work-stealing, since it is the cause of the only communication (and thus dependencies) between processes. Checkpoints resulting from work-stealing are called *forced checkpoints*. Then we will consider the periodic checkpoints, called *local checkpoints*, which are stored periodically, after expiration of pre-defined periods τ.

4.2.1 Forced checkpoints

The TIC protocol with respect to forced checkpoints is defined in Figure 4, showing events A through

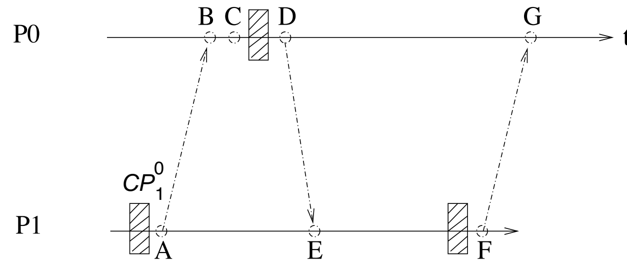

Figure 4. TIC protocol: forced checkpoints.

G for two processes P_0 and P_1. Initially P_0 is executing a task from its stack. The following sequence of events takes place:

1. A process P_1 is created on an idle resource. If it finds a process P_0 that has a potential task to be stolen, it creates a "theft" task T_t charged with stealing a task from process P_0. Before executing T_t, process P_1 checkpoints its state in CP_1^0. Event A is the execution of T_t which sends a *theft request* to P_0.

2. Event B is the receipt of the *theft request* by P_0. Between event B and C it identifies a task T_s and flags it as "stolen by P_1". Between events B and C victim P_0 is in a critical section with respect to theft operations.

3. Between event C and D it forces a checkpoint to reflect the theft. At this time P_0 becomes a victim. Event D constitutes sending T_s to P_1.

4. Event E is the receipt of the stolen task T_s from P_0. Thief P_1 creates entries for two tasks, T_s and T_r, in its stack, as shown in Figure 2. Task T_r is charged with returning the results of the execution of T_s to P_0 and becomes ready when T_s finishes.

5. When P_1 finishes the execution of T_s it takes a checkpoint and executes T_r, which returns the result of T_s to P_0 in event F.

6. Event G is the receipt of the result by P_0.

4.2.2 Local checkpoints

Local checkpoints of each process P_i are stored periodically, after the expiration of the pre-defined period τ. Specifically, after the expiration of τ a process receives a signal to checkpoint. The process can now take a checkpoint. However, there are two exceptions. First, if the process has a task in state *executing* it must wait until execution is finished. Second, if a process is in the critical section between events B and C, checkpointing must be delayed until exiting the critical section. A checkpointing scenario comprising local and forced checkpoints is shown in Figure 5 where local and forced checkpoints are shown unshaded and shaded respectively. Note that the temporal spacing of the two local (unshaded) checkpoints on process P_0 is at least τ.

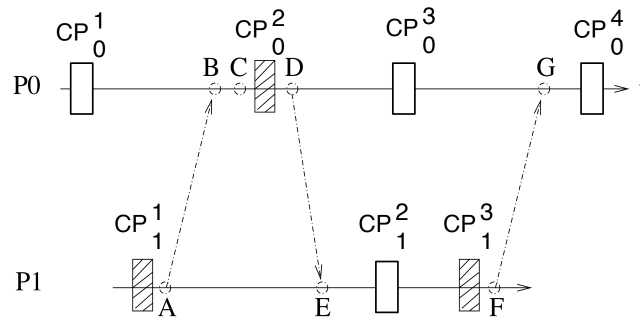

Figure 5. TIC protocol: local and forced checkpoints.

4.2.3 TIC rollback

The objective of TIC is to allow rollback of only crashed processes. A process can be rolled back to its last checkpoint. In fact, for each process only the last checkpoint is kept. We now present a theorem that proves that under TIC a global consistent state of the execution is maintained.

Theorem 1 *Under the TIC protocol the faulty processes can be rolled back, while guaranteeing a consistent global state of the execution.*

Proof: In general, to show that a set of checkpoints form a consistent system state, three conditions must be satisfied [22]: IC1: There is exactly one recovery point for each process. IC2: There is no event for sending a message in a process P *after* its recovery point, whose corresponding receive event in another process Q is *before* the recovery point of Q. IC3: There is no event of sending a message in a process P *before* its recovery point, whose corresponding receive event in another process Q is *after* the recovery point of Q. The scenarios representing conditions IC2 and IC3 are depicted in Figure 6.

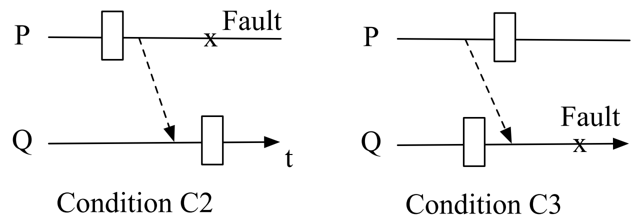

Figure 6. Sources of Inconsistency.

Proving that condition IC1 is met is trivial since TIC stores only the last checkpoint in stable storage. In the remainder of the proof of TIC we will consider all actions possible with respect to the events and checkpoints shown in Figure 5. This enumeration of events and checkpoints is exhaustive.

Part I: Let us assume that processes do not communicate. It is well known that under this assumption a global consistent state of an execution is guaranteed implicitly by using local checkpoints. Thus in the absence of communication only the local process is affected by the rollback. In the context of TIC this means that a process that has not participated in any communication since its last checkpoint, neither as a sender nor receiver, can be rolled back unconditionally to that checkpoint. In Figure 5 this scenario covers, for each checkpoint, the time interval which starts at the time the checkpoint is established until the next event or checkpoint. If $t(CP_i^j)$ denotes the time at which checkpoint CP_i^j is established and $t(X)$ denotes the time of event X, then rollback during the following intervals will maintain a consistent

execution state: $[t(CP_0^1), t(B))$, $[t(CP_0^2), t(D))$, $[t(CP_0^3), t(G))$, $[t(CP_0^4), -)$ for process P_0 and $[t(CP_1^1), t(A))$, $[t(CP_1^2), t([CP_1^3))$, and $[t(CP_1^3), t(F))$ for process P_1. Note that the intervals are open to the right, i.e. the right side of an interval is the time *before* the event. Furthermore, symbol '–' in $[t(CP_0^4), -)$ indicates the time of the next event or checkpoint.

Part II: Now we prove that TIC can deal with rollback that affects or is affected by communication, i.e. we need to show how TIC effectively avoids inconsistency with respect to conditions IC2 and IC3. Recall that the only communication in the system is that due to task stealing, i.e. three communications per theft as shown in Figure 5. An attempt to communicate with a crashed process will result in failure, indicated by an error code generated by the transport layer, e.g., transport control protocol TCP. This error code is used to initiate actions with respect to IC2 and IC3.

We now present systematically, for each of the three communications of TIC, the three possible fault cases as they relate to the treatment of IC2, IC3 and a double fault. The discussion is based on Figure 5.

Communication A to B – the *theft request*:

1) If thief P_1 crashes such that it rolls back past event A, condition IC2 arises. This presents no problem for the new process P_1' (replacing the crashed P_1). P_1' simply requests a theft from another process. P_0 on the other hand will detect the rollback upon unsuccessfully attempting to communicate with the crashed P_1 (in event D), where it receives an error code. P_0 thus voids the theft, i.e., it un-labels task T_s and takes another checkpoint reflecting its new state. Note that this checkpoint is a new version of the checkpoint between C and D.

2) If victim P_0 crashes after event B but before CP_0^2, then condition IC3 is introduced. However, this presents no problem for P_1 who simply times out while waiting for event E. P_1 makes another request.

3) A double fault implies that upon rollback of P_1 as P_1' the re-initiation of event A returns an error. P_1' will inquire about replacement P_0' for the non-responding process P_0. If P_0' has not passed event B, then this constitutes a new theft request. If P_0' has been restarted from CP_0^2 then P_0' will detect that the

thief has also been rolled back upon an unsuccessful event D and will void the theft. This is exactly the action the victim took in case 1).

Communication D to E – the actual theft:

1) If P_0 fails after event D but before it could checkpoint then condition IC2 arises. The (rolled back) victim will initiate another event D to the same thief for the same request (indicated by CP_0^2). This is recognized by P_1 as a duplicate and is ignored.

2) If the thief crashes after the actual theft (event E), but before it was able to checkpoint, then condition IC3 arises. The thief is simply rolled back as P_1' to the initial checkpoint CP_1^0 where it will re-request a task from P_0 (event A). Victim P_0, recognizing the redundant request, changes the state of T_s from *stolen* to *ready*, thus nullifying the old theft, and treats the *theft request* as a new request.

3) The victim is rolled back past event D and finds out the thief does not respond; a double fault. Thus victim P_0' inquires about the replacement process P_1'. If P_1' was initialized with CP_1^1 it will find out about the new P_0' as the result of a communication error at event A. If P_1' was rolled back with a checkpoint taken after event E, then it takes a new CP_0^2 to reflect that P_1' is the rolled back thief.

Communication F to G – the return of the result to the victim:

1) If the thief crashes after event F then condition IC2 arises. Upon re-initiating event E the victim will simply ignore the duplication. Note that this can only occur in the tiny interval after F and before P_1's termination.

2) A crash of the victim after it has received the result (event G) but before it can checkpoint will result in condition IC3. This would stall the victim after rollback to a state where the task is still flagged as stolen, i.e. P_0' would never receive the result in event G. Therefore, as part of the rollback procedure, the victim inspects the last checkpoint for tasks that have been flagged stolen. If the victim's checkpoint contains references to a thief P_1 that is already terminated, it rolls back P_0 on P_0' using the checkpoint of P_0 together with the thief's final checkpoint containing the result. Thus, the rollback uses G_0 and G_1

(which contains only T_r). On the other hand, if the last checkpoint contains references to thieves that are still executing, no action is required since the thief, upon attempting to send the results to the old process P_0, will experience an error from the transport layer and will inquire about P_0'.

3) If the thief is rolled back to CP_1^3 and finds out during event F that the victim has crashed as well, it inquires about P_0'. P_0' will have either been initiated with CP_0^2 or a checkpoint taken after event D, say CP_0^3. In the first case as the result of the error during event D, P_0' enquires about the replacement victim and updates CP_0^2. In the second case it will be waiting for event G, which is coming from the replacement thief. The thief found out about P_0' as a result of the communication error at event F during the attempt to reach the old victim.

Part III: So far we have proven that using TIC inconsistencies are avoided. However, it remains to be established why the three forced checkpoints shown (shaded) in Figure 5 are necessary. Let CP_1^0 and CP_1^f denote the first and final checkpoint of a thief P_1 respectively. The initial checkpoint CP_1^0 guarantees that there exists at least one record of a *theft request* for a thief that crashes. Thus, upon a crash, the thief is rolled back on the new process P_1'. Without CP_1^0 any crash before a checkpoint on the thief would simply erase any reference of the theft (event E), and would stall the victim. The final checkpoint of the thief, CP_1^f, is needed in case the victim P_0 crashes after it has received the results from the thief, but before it could checkpoint its state reflecting the result. Thus, if the victim crashes between event G and its first checkpoint after G, then the actions describing *Communication F to G* will ensure the victim can receive the result of the stolen task.

It should be noted that the final checkpoint of the thief cannot be deleted until the victim has taken a checkpoint after event G, thereby checkpointing the result of the stolen task. Lastly, the forced checkpoint of the victim (between events C and D) ensures that a crash after this checkpoint does not result in the loss of the thief's computation, i.e., there will be a record that allows the victim's replacement process to find the thief. □

The actions described in the proof above constitute a new generation of the protocol, i.e., the concept of a proactive manager, as described in [14, 15], has been eliminated. It has been replaced with a passive name server implemented on the same reliable storage system that facilitates the checkpoint server.

5 Systematic Event Logging

Whereas the TIC protocol was defined with respect to Level 1 of Figure 1, we will now introduce a Level 0 protocol called *Systematic Event Logging* (SEL), which was derived from a log-based method [1]. The motivation for SEL is to reduce the amount of computation that can be lost, which is bound by the execution time of a single failed task[1]. We will later elaborate on the differences between TIC and SEL in their analysis presented in Section 6.

In SEL only the events relevant for the construction of the dataflow graph are logged. Logging events for tasks are their additions and deletions. Logging events of shared data objects are their additions, modifications and deletions. A recovery consists of simply loading and rebuilding subgraph G_i associated with the failed process P_i from the respective log.

The SEL protocol implies the validity of the PWD hypothesis, which was introduced in Subsection 2.1. For the hypothesis to be valid the following two conditions must hold:

- C_1: Once a task starts executing it will continue, without being affected by external events, until its execution ends.
- C_2: The execution of a task is deterministic with respect to the tasks and shared data objects that are created. Note that this implies that the execution will always create the same (isomorphic) dataflow graph.

At first sight condition C_1 may appear rather restrictive. However, this is not the case for our application domain, i.e., large parallel executions, (see Equation 1 below).

If all tasks of a dataflow graph obey conditions C_1 and C_2, then all processes executing the graph will comply with the PWD hypothesis. The idea behind the proof of this theorem is simple. In the

[1]Recall that the task is the smallest unit of execution in the execution model.

execution model, the execution of tasks is deterministic, whereas the starting time of their execution is non-deterministic. However, this implies, in turn, that during the execution of a task in the execution model, it itself will create the same sequence of tasks and data objects.

In case of a fault, task duplication needs to be avoided during rollback. Specifically, in the implementation one has to guarantee that only one instance of a any given task can exist. In the absence of such guarantee, it could happen that during rollback a task recreates other tasks or data objects that already exist from earlier failed executions. Note that, depending on the timing of the fault, this could result in a significant number of duplicated nodes, since each duplicated task itself may be the initiator of a significant portion of computation. In our implementation of SEL duplication avoidance is achieved using a unique and reproducible identification method of all vertices in the graph.

6 Complexity Analysis

In this section we present a cost model for the TIC and SEL protocol. But first we want to introduce the necessary notation and analyze the general work-stealing model.

Let T_{sec} be the time of execution of a sequential program on a single processor. Furthermore, let T_1 denote the time of the execution of the corresponding parallel program on a single processor and let T_∞ be the theoretical execution time of the application as executed on an unbounded number of processors. Thus T_∞ represents the execution time associated with the critical-path. It should be noted that in large executions suitable for parallel environments we always have

$$T_1 \gg T_\infty. \tag{1}$$

Next, let T_p be the execution time of a program on p identical physical processors. Then the execution of a parallel program using work-stealing is bound by [11]

$$T_p \leq \frac{T_1}{p} + c_\infty T_\infty \tag{2}$$

where constant c_∞ defines a bound on the overhead associated with the critical-path, including the scheduling overhead. Furthermore, we have

$$T_1 \leq c_1 T_{sec} \tag{3}$$

where c_1 corresponds to the maximum overhead induced by parallelism, excluding the cost of scheduling. The constants c_1 and c_∞ depend on the specific implementation of the execution model and are a measure of the implementation's efficiency.

To show how little impact the term $c_\infty T_\infty$ of Equation 2 has, one should note that the number of thefts performed by any process[2], denoted by N_{theft}, which introduce the scheduling overhead hidden in c_∞ is small [11, 12], since

$$N_{theft} \leq O(T_\infty). \tag{4}$$

Specifically, with $T_1 \gg T_\infty$ we can approximate Equation 2 by $T_p \approx \frac{T_1}{p}$.

6.1 Analysis of Fault-free Execution

If we add a checkpointing mechanism, it is of special interest to analyze its overhead associated with fault-free execution, since the occurrence of faults is considered to be the rare exception rather than the norm.

6.1.1 Analysis of TIC

In TIC, a checkpoint is performed (1) periodically for each process, as dictated by period τ, and (2) as the result of work-stealing. Let T_P^{TIC} denote the execution of a parallel program on p processors under TIC. Then,

$$T_P^{TIC} \leq T_p + \max_{i=1,\ldots,p} \{Overhead_i^{TIC}\}, \tag{5}$$

where $Overhead_i^{TIC}$ denotes the total TIC checkpointing overhead on processor P_i. This overhead depends on the total number of checkpoints taken on processor P_i and the overhead of a single checkpoint. The maximal number of checkpoints performed by a processor is $[T_P^{TIC}/\tau + O(N_{theft})]$, where T_P^{TIC}/τ indicates the number of checkpoints due to period τ and N_{theft} is the maximal number of thefts performed by any processor. Note that we use $O(N_{theft})$, since with respect to Figure 4 the number of checkpoints of the thief and the victim are not equal.

[2] We assume that at any given time at most one process is active on a processor.

The overhead of a single checkpoint in TIC is associated with storing the collection of vertices in G_i and depends on two parameters. First, it depends on the size of G. Specifically, it depends on the number of tasks and shared data objects, as well as the size of the latter. Second, it depends on the time of an elementary access to stable storage, denoted by t_s.

The number of vertices in G_i has an upper bound of N_∞, which denotes the maximum number of vertices in a path of G [11]. The checkpoint overhead for processor P_i is thus bound by

$$Overhead_i^{TIC} = [T_P^{TIC}/\tau + O(N_{theft})]\, f_{overhead}^{TIC}(N_\infty, t_s). \tag{6}$$

The function $f_{overhead}^{TIC}()$ indicates the overhead associated with a single checkpoint and depends only on G, or more preceisly N_∞, as well as t_s.

6.1.2 Analysis of SEL

As defined in Section 5, in SEL a log is performed for each of the described events relevant for the construction of G, i.e., (1) vertex creation, (2) shared data modification and (3) vertex deletion. Recall that in $G = (\mathcal{V}, \mathcal{E})$ a vertex $v_i \in \mathcal{V}$ is either a task or a shared data object.

Let T_P^{SEL} denote the execution of a parallel program on p processors under SEL. Then T_P^{SEL} can be expressed as

$$T_P^{SEL} \leq T_p + \max_{i=1,\ldots,p}\{Overhead_i^{SEL}\}. \tag{7}$$

This overhead depends on the total number of vertices in G_i and the overhead of a single event log. The maximal number of logs performed by a processor is $|G_i|$, i.e., the number of vertices in G_i.

The overhead of a single event log in SEL is associated with storing a single vertex v_j of G_i and depends on two parameters. Specifically, it depends on the size of v_j and the access time to stable storage t_s. Note that if v_j is a task, then the log is potentially very small and of constant size, whereas if it is a data object, then the log size is equal to that of the object. The logging overhead for processor P_i is thus bound by

$$Overhead_i^{SEL} = |G_i| f_{overhead}^{SEL}(|v_j|, t_s). \tag{8}$$

The function $f_{overhead}^{SEL}()$ indicates the overhead associated with a single log.

6.2 Analysis of Executions Containing Faults

The overhead associated with fault-free execution is the penalty one pays for having a recovery mechanism. It remains to be shown how much overhead is associated with recovery as the result of a fault and how much execution time can be lost under different strategies.

The overhead associated with recovery is due to loading and rebuilding the affected portions of G. This can be effectively achieved by regenerating G_i of the affected processes. Thus, the time of recovery of a single process P_i, denoted by $t_i^{recovery}$, depends only on the size of its associated subgraph G_i, i.e., $t_i^{recovery} = O(|G_i|)$. Note that for a global recovery, as the result of the failure of the entire application, this translates to $max(t_i^{recovery})$ and not to $\sum t_i^{recovery}$.

The way G_i is rebuilt for a failed process differs for the two protocols. Under TIC rebuilding G_i implies simply reading the structure from the checkpoint. For SEL this is somewhat more involved, since now G_i has to be reconstructed from the individual logs.

Next, we address the amount of work that a process can lose due to a single fault. In TIC this is the maximal difference in time between two consecutive checkpoints. This time is defined by the checkpointing period τ and the execution time of a task, since a checkpoint of a process that is executing a task cannot be made until the task finishes execution. In the worst case, the process receives a checkpointing signal after τ and has to wait for the end of the execution of its current task before checkpointing. Thus, the time between checkpoints is bound by $\tau + \max(c_i)$ where c_i is the computation time of task T_i. But how bad can the impact of c_i be? In a parallel application it is reasonable to assume $T_\infty \ll T_1$. Since T_∞ is the critical path of the application any $c_i \leq T_\infty$. As a result one can assume c_i to be relatively small.

In SEL, due to its fine granularity of logging, the maximum amount of execution time lost is simply that of a single task. However, this comes at the cost of higher logging overhead, as was addressed in Equation 8.

6.3 Discussion

The overhead of the TIC protocol depends on the number of theft operations and period τ. To reduce the overhead, one needs to increase τ. However, this also increases the maximum amount of computation that can be lost.

For SEL the overhead depends only on the size of graph G, i.e., its vertices v_i which have to be saved. If one wants to reduce the overhead one has to reduce the size of G. This however reduces the parallelism of the application.

Comparing the TIC and SEL protocol makes only sense under consideration of the application, e.g., number of tasks, task size or parallelism. If $T_\infty \ll T_1$, given a reasonable value[3] for τ, then the overhead of TIC is likely to be much lower than that of SEL, i.e., given Equations 6 and 8, $[T_P^{TIC}/\tau + O(N_{theft})]$ is most likely much smaller than $|G_i|$, thus more than compensating for $f_{overhead}^{SEL}(|v_j|, t_s) < f_{overhead}^{TIC}(N_\infty, t_s)$, as will be confirmed by the results in Section 7. The reduced overhead has huge implication on the avoidance of bottlenecks in the checkpointing server(s). For applications with large data manipulations, TIC, with an appropriate choice of τ, may be the only choice capable of eliminating storage bottlenecks.

On the other hand, SEL addresses the needs of applications with low tolerance for lost execution time. However, one has to analyze the bandwidth requirements of logging in order to determine feasibility.

It should be emphasized that the advantage of the TIC and SEL protocols is that they do not require replacement resources for failed processes, e.g., the failed process can be rolled back on an existing resource. This is due to the fact that the state of the execution is platform and configuration independent.

Lastly, we want to indicate that, even though the TIC protocol has been motivated by Communication Induced Checkpointing (CIC) [3], TIC has multiple advantages over CIC. First, unlike CIC, in TIC only the last checkpoint needs to be kept in the stable storage. This has potentially large implications on the amount of data that needs to be stored. Thus, the advantage of TIC is the reduction of checkpointing data as well as the time it takes to recover this data during roll-back. The second significant advantage

[3]Note that unreasonably small values of τ would result in excessive local checkpointing.

is that in TIC only the failed process needs to be rolled back. Note that in CIC all processes must be rolled back after a fault.

7 Experimental Results

7.1 Application and platform description

The performance and overhead of the TIC and SEL protocols were experimentally determined for the *Quadratic Assignment Problem* (instance[4] NUGENT 22) which was parallelized in KAAPI. The local experiments were conducted on the iCluster2[5], which consists of 104 nodes interconnected by a 100Mbps Ethernet network, each node featuring two Itanium-2 processors (900 MHz) and 3 GB of local memory. The inter-cluster experiments were conducted on Grid5000 [13], which consists of clusters located at nine French institutions.

In order to take advantage of the distributed fashion of the checkpoint, i.e., G_i, each processor has a dedicated checkpoint server. This configuration has two advantages. First, it reflects the theoretical assumptions of Section 6 and second, the actual overhead of the checkpointing mechanism is measured, rather than the overhead associated with a centralized checkpoint server.

7.2 Fault-free Executions

We will now investigate the overhead of the protocols in fault-free executions, followed by executions containing faults. Then we show the results of a real-world example executing on heterogeneous and dynamic cluster configurations. We conclude with a comparison of both protocols with the closest counterpart, i.e., Satin [31].

The impact of the degree of parallelism can be seen in Figure 7, where the number of parallel tasks generated during execution grows as the size of tasks are reduced. Recall that the number of tasks directly relates to the size of graph G, which in turn has implication with respect to the overhead of the

[4] see http://www.opt.math.tu-graz.ac.at/qaplib/

[5] http://www.inrialpes.fr/sed/i-cluster2

protocols. The degree of parallelism increases drastically for threshold 5 and approaches its maximum at threshold 10.

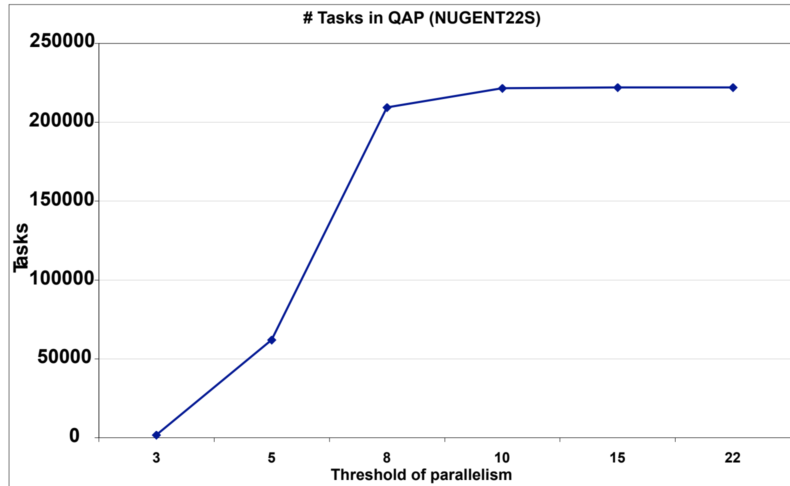

Figure 7. Tasks and Application Granularity.

Figure 8 shows the execution times of the application for different protocols in the absence of faults. Two observations can be made. First, the application scales with the number of processors for all protocols. Second, there is very little difference between the execution times of the protocols for the same number of processors. In fact, the largest difference among the executions was observed in the case of 120 processors and was measured at 7.6%. It is easy to falsely conclude that, based on the small differences shown in the scenarios of Figure 8, all protocols perform approximately the same. The important measure of overhead of the mechanism is the total amount of data associated with the protocol that is sent to stable storage. This overhead is affected by the total size and the number of messages. Due to the efficient, distributed configuration of the experiment, which may not be realistic for real-world applications, this overhead was hidden and thus does not show in the figure. Figure 9 addresses this cost, i.e., the cost of the fault-tolerance mechanism that the infrastructure has to absorb, and shows the total volume of checkpointing and logging data stored. The advantages of TIC can be seen in the significant

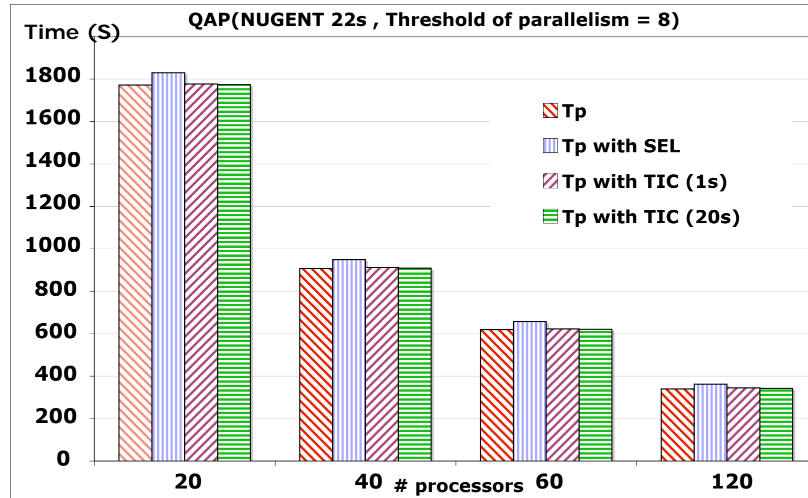

Figure 8. Execution Times of Protocols.

reduction of data, which is most visible for larger periods τ. Furthermore, the data volume stays relatively constant for different number of processors. This is due to the fact that the number of thefts, and thus theft-induced overhead, is actually very small, as was explained in Section 6.

7.3 Executions with Faults

To show the overhead of the mechanisms in the presence of faults, we consider executions containing faults. First, we want to measure the cost induced by the computation lost due to the fault(s) and the overhead of the protocols. Specifically, for each protocol we show

$$\frac{T_p^{withfault} - T_p'}{T_p'} \tag{9}$$

where $T_p^{withfault}$ is the time of execution in the presence of faults and roll-back, and T_p' is the time of a fault-free execution.

Figure 10 shows the measured cost using Equation 9 for different numbers of faults. The interpretation of T_p' is the execution time of the application including the overhead of the checkpointing or logging

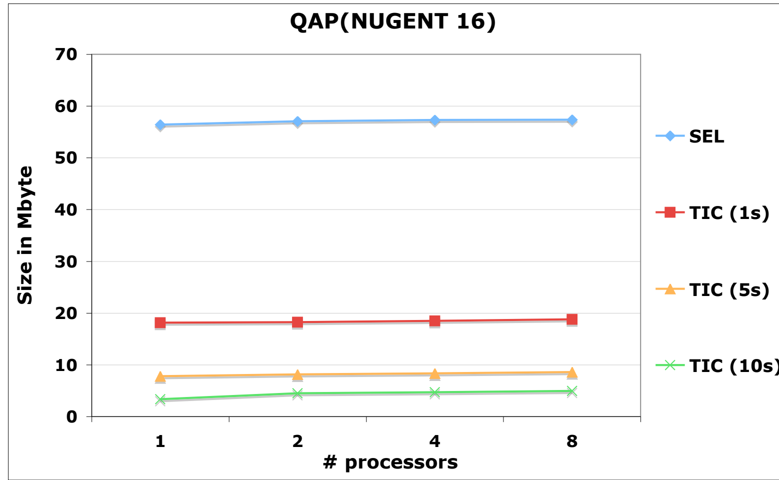

Figure 9. Total Volume of Data Stored.

mechanism. One can observe that, as the number of faults increase, the execution time grows linearly. Note that, since the overhead of the protocols is included in T'_p, the values displayed are the computation time lost due to the faults as well as the overhead of roll-back, but do not contain the overhead of checkpointing or logging. As expected, and discussed in Subsection 6.3, the computation lost using SEL is lower than that under TIC, since in SEL only the computation of failed tasks are lost. For the experiment the period in TIC was set at $\tau = 1s$ and the mean task execution time was 0.23s.

However, Figure 10, with its interpretation of T'_p, does not account for the overhead of checkpointing or logging. This overhead was included in the measurement shown in Figure 11. Now T'_p in Equation 9 is the execution time of the application without any fault-tolerance protocol, i.e., neither SEL nor TIC. The measurements reveal that the actual overhead of SEL overshadows its advantages shown in Figure 10. Specifically, accounting for the overhead of checkpointing (of TIC) and logging (of SEL), the real advantage of lower checkpointing overhead of TIC surfaces.

7.4 Application Executing on Heterogeneous and Dynamic Grid

Next we show an application of TIC in a heterogeneous Grid. Four clusters of Grid5000 (geographically dispersed in France) were used, utilizing different hardware architectures. The execution clusters

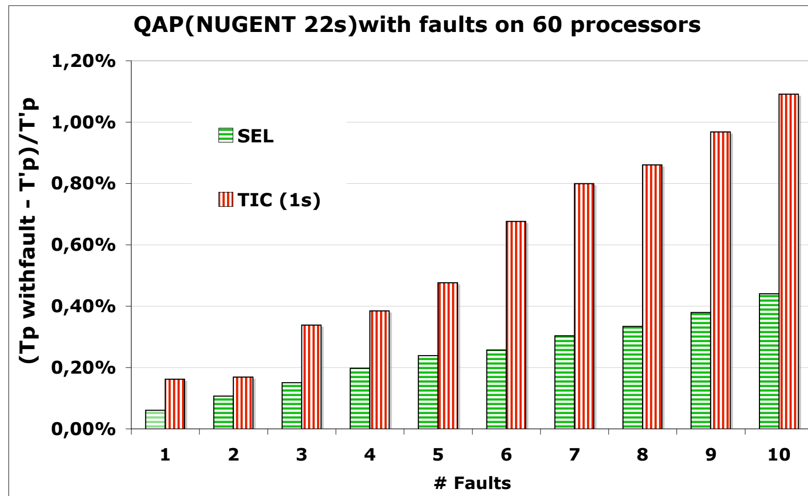

Figure 10. Overhead of Rollback.

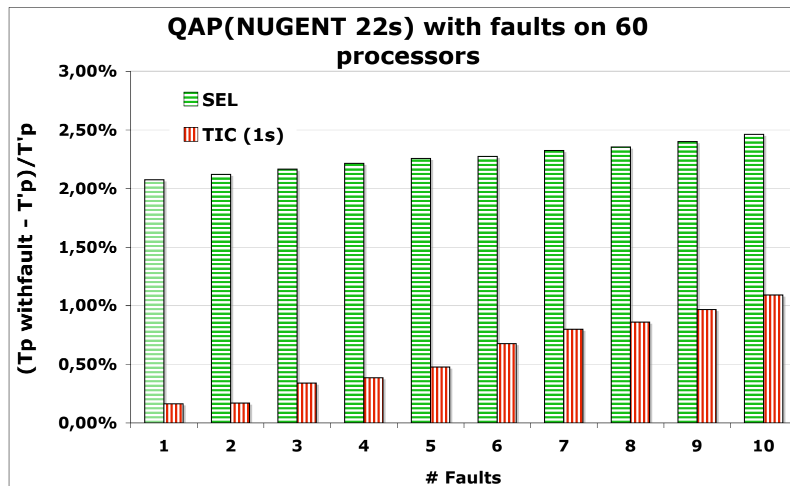

Figure 11. Total Overhead considering Faults.

used AMD Opteron, Intel Xeon and PowerPC architectures respectively, whereas the stable storage cluster used Xeons.

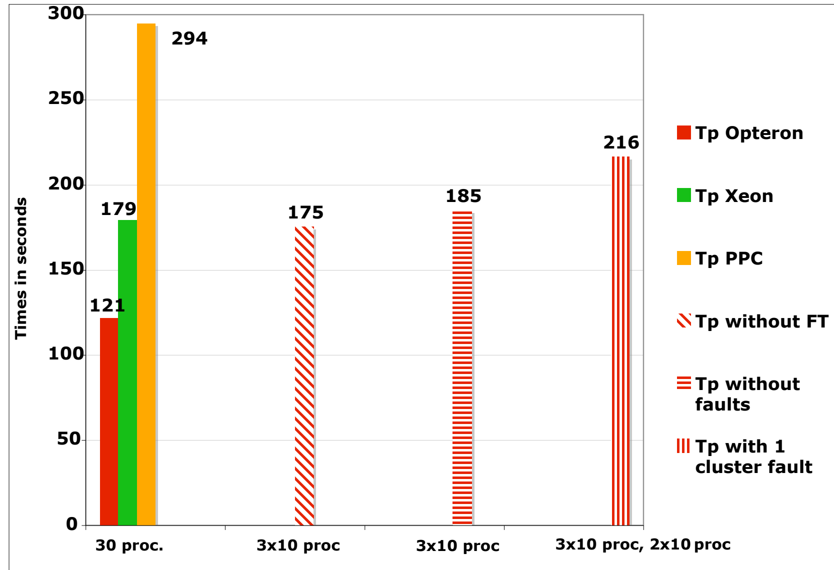

Figure 12. QAP Application on Grid5000.

Figure 12 summarizes several experiments. First, the entire application was executed on each of the three execution clusters using 30 computational nodes. The respective execution times are shown in the three bars to the left.

Next, the application was executed on all three execution clusters, using 10 nodes on each cluster. Thus the total number of processors available to the application was again 30. The fourth bar of Figure 12 shows the time of the fault-free execution (175 seconds) using no fault-tolerance protocol at all. Next, the same experiment was repeated using the TIC protocol with $\tau = 5s$. The result is shown in the fifth bar peaked at 185 seconds. The difference in execution times between this and the previous scenario is entirely due to the overhead of TIC and its remote checkpointing. Finally, an execution with fault in the PowerPC cluster was considered. Specifically, after 50% of the application had executed a fault was injected that affected all 10 nodes of the PowerPC cluster, i.e., the cluster was lost. The affected part of the execution rolled back and finished execution on the remaining 20 processors. One can see (in the

bar to the right indicating 216 seconds) that the execution tolerated the cluster fault exceptionally well, resulting in an overall execution time which was only 17% larger than that of the fault-free case, even though one entire cluster was permanently lost. Furthermore, the rollback was across platforms, i.e., the computations of the failed cluster was dynamically absorbed by the two remaining clusters using different hardware architectures.

7.5 Comparison with Satin

A fault-tolerant parallel programming environment similar to the approach presented above is Satin [31]. In fact, the Satin environment follows the general execution model presented in Figure 1. However, the abstraction of the execution state is a series-parallel graph, rather than the dataflow graph. As such Satin only addresses recursive, series-parallel programming applications. In Satin fault-tolerance is based on redoing the work lost by the crashed processor(s). To avoid redundant computations, partial results, which are stored in a global replicated table, can later be reused during recovery after a crash.

To compare the performance of TIC with Satin a different application was used, i.e., a recursive application resembling a generalization of a Fibonacci computation. Figure 13 shows the result of executions of both approaches for different fault scenarios. Specifically, for each approach first an execution without fault is shown. Next a single fault was injected after 25%, 50% and 75% of the execution had completed. To eliminate the impact of the different implementation languages and execution environments on the execution times, i.e., C++/KAAPI and Java/Satin, the measurements presented in the figure are relative to the execution times in their respective environments. As can be seen, the cost in Satin is significantly higher than that in KAAPI/TIC, which used $\tau = 1s$. The reason is that in Satin all computations affected by the fault are lost. In fact, the loss is higher the later the fault occurs during the execution. This is not the case in TIC where the maximum loss is small, i.e., $\tau + \max(c_i)$ as was shown in Subsection 6.2. Thus TIC overcomes this performance deficiency of Satin.

On the other hand, the TIC protocol is pessimistic in the sense that processes are always checkpointed to anticipate a future failure. The result is that for fault-free executions the Sating approach has lower

overhead than TIC. However, as was shown in Subsection 7.2, the overhead of TIC is very small.

For applications with small computation times (linear or quasi linear) Satin also tends to perform better than TIC. The reason is that the time to recompute solutions under Satin may be less than the overhead associated with writing checkpoints to stable storage. However, such applications are difficult to parallelize due to the low computation/communication ratio.

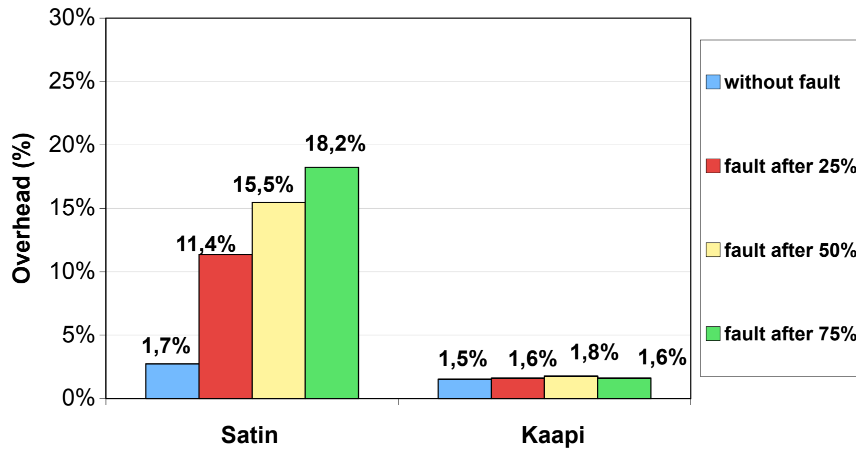

Figure 13. Comparison of Satin with KAAPI/TIC using 32 Processors.

8 Conclusions

To overcome the problem of applications executing in large systems where the MTTF approaches or sinks below the execution time of the application, two fault-tolerant protocols, TIC and SEL, were introduced. The two protocols take under consideration the heterogeneous and dynamic characteristics of Grid or cluster applications that pose limitations on the effective exploitation of the underlying infrastructure. The flexibility of dataflow graphs has been exploited to allow for a platform-independent description of the execution state. This description resulted in flexible and portable rollback recovery strategies.

SEL allowed for rollback at the lowest level of granularity, with a maximal computational loss of one task. However, its overhead was sensitive to the size of the associated dataflow graph. TIC experienced

lower overhead, related to work-stealing, which was shown bounded by the critical path of the graph. By selecting an appropriate application granularity for SEL and period τ for TIC the protocols can be tuned to the specific requirements or needs of the application. A cost model was derived, quantifying the induced overhead of both protocols. The experimental results confirmed the theoretical analysis and demonstrated the low overhead of both approaches.

9 Acknowledgements

The authors wish to thank Jean-Louis Roch, ID-IMAG, France, for all the discussions and valuable insight that lead to the success of this research.

References

[1] L. Alvisi, and K. Marzullo, *Message Logging: Pessimistic, Optimistic, Causal and Optimal*, IEEE Transactions on Software Engineering, Vol. 24, No. 2, pp. 149-159, 1998.

[2] K. Anstreicher, N. Brixius, J-P. Goux and J. Linderoth, *Solving large quadratic assignment problems on computational grids*, Mathematical Programming, Vol. 91, No. 3, 2002.

[3] R. Baldoni, *A Communication-Induced Checkpointing Protocol that Ensures Rollback-Dependency Trackability*, Proc. 27th Intl. Symposium on Fault-Tolerant Computing (FTCS '97), p. 68, 1997.

[4] F. Baude, D. Caromel, C. Delb and L. Henrio, *A Hybrid Message Logging-CIC Protocol for Constrained Checkpointability*, Proc. EuroPar2005, LNCS, Springer Verlag, pp. 644-653, 2005.,

[5] G. Bosilca et.al., *MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes*, Proc. SuperComputing, Nov., 2002.

[6] A. Bouteiller et.al., *MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on the Pessimistic Sender Based Message Logging*, Proc. 2003 ACM/IEEE Conference on Supercomputing, (SC'03), pp. 1-17, 2003.

[7] A. Bouteiller, P. Lemarinier, G. Krawezik and F. Cappello, *Coordinated checkpoint versus message log for fault tolerant MPI*, Proc. 2003 IEEE International Conference on Cluster Computing, Honk Hong, p. 242, 2003.

[8] S. Chakravorty and L. V. Kale, *A Fault Tolerant Protocol for Massively Parallel Machines*, FTPDS Workshop for the 18th Intl. Parallel and Distributed Processing Symposium (IPDPS'04), pages 212a, 2004.

[9] K. M. Chandy and L. Lamport, *Distributed snapshots: determining global states of distributed systems*, ACM Trans. Computer Systems, Vol. 3, No. 1, pp. 63-75, 1985.

[10] E. N. Elnozahy, L Alvisi, Y-M. Wang and D.B. Johnson, *A survey of rollback-recovery protocols in message-passing systems*, ACM Computing Surveys, Vol. 34, Issue 3, pp. 375-408, Sept., 2002.

[11] M. Frigo, C. E. Leiserson, and K. H. Randall, *The implementation of the Cilk-5 multithreaded language*, Proc. ACM SIGPLAN 1998 conference on Programming language design and implementation, pp. 212-223, 1998.

[12] F. Galilée, J-L. Roch, G. Cavalheiro and Doreille, *Athapascan-1: On-line Building Data Flow Graph in a Parallel Language*, Proc. IEEE Annual Conference on Parallel Architectures and Compilation Techniques, PACT'98, pp. 88-95, 1998.

[13] Grid5000, A large scale nation-wide infrastructure for Grid research, France, https://www.grid5000.fr, 2006.

[14] S. Jafar, A. Krings, T. Gautier and J-L. Roch, *Theft-Induced Checkpointing for Reconfigurable Dataflow Applications*, Proc. IEEE Electro/Information Technology Conference, EIT-2005, May 22-25, Lincoln, Nebraska, (6 pages), 2005.

[15] S. Jafar, T. Gautier, A. Krings and J-L. Roch, *A Checkpoint/Recovery Model for Heterogeneous Dataflow Computations Using Work-Stealing*, Lecture Notes Computer Science (LNCS), Proc. Euro-Par 2005, August 30 - Sept. 2, Lisboa, Portugal, pp.675-684, 2005.

[16] A. W. Krings, J-L. Roch, S. Jafar and S. Varrette, *A Probabilistic Approach for Task and Result Certification of Large-scale Distributed Applications in Hostile Environments*, European Grid Conference (EGC2005), February 14-16, in LNCS 3470, editors P. Sloot et.al., (12 pages), 2005.

[17] A. W. Krings, J-L Roch and S. Jafar, *Certification of Large Distributed Computations with Task Dependencies in Hostile Environments*, Proc. IEEE Electro/Information Technology Conference, EIT-2005, May 22-25, Lincoln, Nebraska, (6 pages), 2005.

[18] L. Lamport, M. Pease, R. Shostak, *The Byzantine Generals Problem*, ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, pp. 382-401, July 1982.

[19] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, Tech. Report CS-TR-97-1346, Univ. Wisconsin, Madison, 1997.

[20] A. Nguyen-Tuong, A. Grimshaw, and M. Hyett, *Exploiting Data-Flow for Fault-Tolerance in a Wide-Area Parallel System*, 15th Symposium on Reliable Distributed Systesm, pp. 2-11, 1996.

[21] D. A. Patterson, G. Gibson and R. H. Katz, *A case for redundant arrays of inexpensive disks (RAID)*, Proc. 1988 ACM SIGMOD international conference on Management of data, pp. 109-116, 1988.

[22] D.K. Pradhan, *Fault-Toerant Computer System Design*, Prentice Hall, 1996.

[23] B. Randell, *System structure for software fault tolerance*, Proc. International Conference on Reliable Software, pp. 437-449, 1975.

[24] L. Sarmenta, *Sabotage-Tolerance Mechanisms for Volunteer Computing Systems*, Future Generation Computer Systems, No. 4, Vol. 18, 2002.

[25] J. Silc, B. Robic and T. Ungerer, *Asynchrony in parallel computing: from dataflow to multithreading*, Progress in Computer Research, pp. 1-33, 2001.

[26] G. Stellner, *CoCheck: Checkpointing and Process Migration for MPI*, Proc. 10th Intl. Parallel Processing Symposium (IPPS'96), 15-19 April, Honolulu, Hawaii, pp. 526-531, 1996.

[27] R. Strom and S. Yemini, *Optimistic recovery in distributed systems*, ACM Transactions in Computer Systems, Vol. 3, No. 3, pp. 204-226, 1985.

[28] V. Strumpen, *Portable and Fault-Tolerant Software Systems*, IEEE Micro, Vol. 18 , Issue 5, pp. 22-32, Sept., 1998.

[29] P. Thambidurai, and Y.-K. Park, *Interactive Consistency with Multiple Failure Modes*, Proc. 7th Symp. on Reliable Distributed Systems, Columbus, OH, pp. 93-100, Oct. 1988.

[30] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*, John Wiley and Sons, New York, 2001.

[31] G. Wrzesinska, R. van Nieuwpoort, J. Maassen and H. E. Bal, *Fault-Tolerance, Malleability and Migration for Divide-and-Conquer Applications on the Grid*, Proc. of 19th International Parallel and Distributed Processing Symposium, p. 13a, April, 2005.

[32] Jay J. Wylie, et.al., *Selecting the Right Data Distribution Scheme for a Survivable Storage System*, Technical Report, CMU-CS-01-120, Carnegie Mellon University, May 2001.

[33] G. Zheng and L. Shi and L. V. Kalé, *FTC-Charm++: An In-Memory Checkpoint-Based Fault Tolerant Runtime for Charm++ and MPI*, Proc. IEEE International Conference on Cluster Computing, San Dieago, Sept. 20-23, pp. 93-103, 2004.