In D. R. Avresky, editor, *Fault-Tolerant Parallel and Distributed Systems*. IEEE Computer Society Press, 1994.

Fault-Tolerant Parallel Programming with Atomic Actions

Joachim Maier
Institute of Parallel and Distributed High-Performance Systems (IPVR)
University of Stuttgart
joachim.maier@informatik.uni-stuttgart.de

Abstract

*The Pact (*parallel* act*ions) parallel programming environment provides an easy-to-use parallel execution and synchronization model based on task parallelization. To give the programmer an abstraction for global data (even on distributed memory machines) the Pact runtime system uses virtual shared memory. Execution's efficiency is improved with data-dependent dynamic load balancing and latency-masking by multithreaded servers. Fault tolerance in Pact is based on atomic actions and is guaranteed by the runtime system in a fully user-transparent way. This article describes the Pact runtime system's design together with its logging and recovery algorithms for an implementation on a massively parallel distributed memory computer.*

1 Introduction

Although parallelism can be used to build fault-tolerant computers (as in [1] and [4]), most parallel computers and their operating systems are not designed for this purpose. The design goal is high performance, but this often causes relatively low reliability [17]. Unfortunately, most parallel computers do not provide mechanisms for fault-tolerant execution of application programs. Therefore, the programmer has to deal with faults by himself. The easiest way to do this is by starting the program once again after a system failure occurs. This means that all work done before the crash is lost. If this is not acceptable, and for long-running applications it definitely is not, the program has to deal with fault tolerance, for example, by using *checkpoints*. In the absence of a checkpointing library this can be done by synchronizing all processes at a consistent global state and writing the results of the computations made up to this time in a checkpoint file. If the system fails after the checkpoint is completed, the program can be restarted by the programmer from the most recently saved state. The problem here is that the programmer has to decide which data has to be written in the checkpoint file and how to recover from this data. This is an unsatisfying solution, because all work has to be done by the programmer, which increases the complexity of the parallel program. To avoid these disadvantages a user-transparent checkpointing library should be used which implements one of the numerous checkpointing protocols for parallel and distributed computer systems described in the literature.

These checkpointing schemes can be broadly classified into *pessimistic* and *optimistic* schemes [13]. Pessimistic schemes, as presented in [2] and [18], log each message synchronously as it is sent or received. This makes recovery fast, but the overhead of synchronously logging every message makes these schemes too expensive for applications on parallel computers where frequent message transfers occur. These schemes are named "pessimistic" because they assume high failure probability, as is typical for distributed computing, and therefore optimize recovery on the cost of high overhead during normal processing. Optimistic schemes, such as those presented in [11] and [20], are based on asynchronous logging of messages. This causes low overhead on normal processing and allows the processes to choose the time to checkpoint their internal state and their messages. This is achieved by a complex recovery algorithm. Recovery requires global rollback to a saved state and subsequent redo of logged messages to restore a consistent global state. Hence, recovery is much more expensive than in pessimistic schemes. The name "optimistic" comes from the optimistic assumption that failures are rare. Recent approaches such as the protocols proposed in [3], [5], and [13] optimize the drawbacks of checkpointing by periodically checkpointing consistent states of the program's processes instead of checkpointing all messages. If this is done with further optimizations like *copy-on-write* and *incremental checkpointing* [6], the runtime overhead is low.

For database systems fault tolerance is guaranteed in a fully transparent way by using *transactions* [9]. Transactions are atomic units of execution, which means that either the whole transaction runs successfully or, if a fault occurs during its execution, all changes on the shared database are undone. This leaves the database in a *consistent* state, even if a system fault occurs. Transaction processing systems have been the subject of studies over decades in various implementations for single- and multiprocessor-systems and for distributed systems. Refer to [10] for a detailed description of the current state of transaction processing. Transactions are not designed for parallel programming, but for database applications, where a user's program should run in an *atomic* (enables fault tolerance due to clear fault semantics) and *isolated* (avoids multi-user anomalies) way on shared databases. All changes of the transaction have to be *durable* after the transaction's *commit* and the database should remain in a *consistent* state under all circumstances. While atomicity and consistency are also desirable for parallel programming, isolation is not advantageous and durability is unnecessary. For this reason transactions are not useful for parallel programming without any modifications. Some work in this direction is described in [14] and [21] where *atomic actions* are used for the implementation of object-oriented distributed systems with durable objects. Because transactions are independent units of execution, there is usually no need for a control structure between them. For parallel programming there is an essential need for a flexible, easy-to-use way to control the program's parallel execution.

In Pact, atomic actions are used to achieve fully user-transparent fault tolerance with low runtime overhead. This is done with the following ideas in mind:

- Using the long-standing experience in transaction processing systems for parallel programming, it is possible to develop an efficient approach to fault-tolerant parallel computing.
- With atomic actions it is possible not only to build efficient restart recovery, but also to build systems with continuous execution by tolerating partial failures.
- Among the main advantages of atomic actions is not only the possible tolerance of system failures, but also flexible reaction and recovery of faults inside the application program because of the simple fault semantics due to atomicity.
- Using atomic actions avoids the *domino effect* (rollback of *all* processes to a globally consistent state) during recovery because recovery of each action is independent of the execution of other actions.

The next section gives a brief description of Pact's programming model. The architecture of Pact's runtime system, the execution of actions, and the recovery algorithms are explained in Section 3. The last two sections describe the experiences with the current prototype running on a Tandem™ multiprocessor system and give some conclusions.

2 Pact's programming model

Since the topic of this paper is the design of the runtime system, Pact's programming model is described only as far as it is necessary to understand the basic ideas. The programming model and Pact's extensions to procedural programming languages are described in more detail in [15]. Because Pact is designed for MIMD computers, it is based on the task parallelization paradigm. Tasks in Pact are called *actions* to reflect their special semantics. The most important property of actions is their *atomicity*. This property guarantees that if an action is executed successfully, its results and its changes on shared data become visible for subsequent actions, or otherwise if an error happens inside of an action the action returns without visible changes on shared data. This makes it easy to handle errors. If, for example, the action gets a division-by-zero error, it is possible just to change the data in an external error handling action and to start the action once again. As reported in [19], using atomic actions increases the quality and reusability of (especially large) code and reduces the complexity by a remarkable amount.

Each parallel action has access to shared data. Contrary to the *strict consistency* used in most shared-memory multiprocessors, where each parallel process has uncontrolled access to the shared data, Pact's actions use an access-dependent two-phase locking scheme. This helps to avert synchronization faults which are often hard to find. The shared data model in Pact is used to provide a high programming abstraction and to enable dynamic load-balancing. To allow efficient access to the shared data and to avoid lock conflicts, the programmer should define *dataunits*, which are either simple shared variables, records, or parts of the large shared variables like vectors and matrices (for example, rows of a matrix for a gaussian elimination algorithm, or submatrices for a block-tridiagonal solver). Dataunits are used as the unit of concurrency control and consistency.

The execution order of parallel actions is defined in *execution plans*, which are made of actions and *events*. Each action is connected to three events: its *start-event*, its *termination-event* and its *error-event*. When the start-event occurs, the action is scheduled by the runtime system. If the action has successfully terminated its execution, the termination-event occurs, which may start the execution of succeeding actions (by triggering their start-events). Error-events give the programmer an elegant possibility to handle software faults. A simple example for an execution plan showing only start- and termination-events in the form of a dependency graph is illustrated on the left-hand side in Figure 1. In this example, the actions g(i), g(i-1), g(i-2), f(i), and f(x) are executed in parallel after the termination of s() – the event ParBegin has been triggered. The action e(y) is started after the event ParEnd has been triggered. This is when all the parallel actions have been terminated. The example on the right-hand side shows error handling with error-events. If the action f(x,y) fails because of an internal error, the error-event error is triggered instead of the termination-event term. After the error-handling in the action errhdlr() has been done, the action f(x,y) can be executed again by triggering the event redo. Redo is possible because in the error case all changes of action f(x,y) on shared data have been undone by the runtime system. If the execution

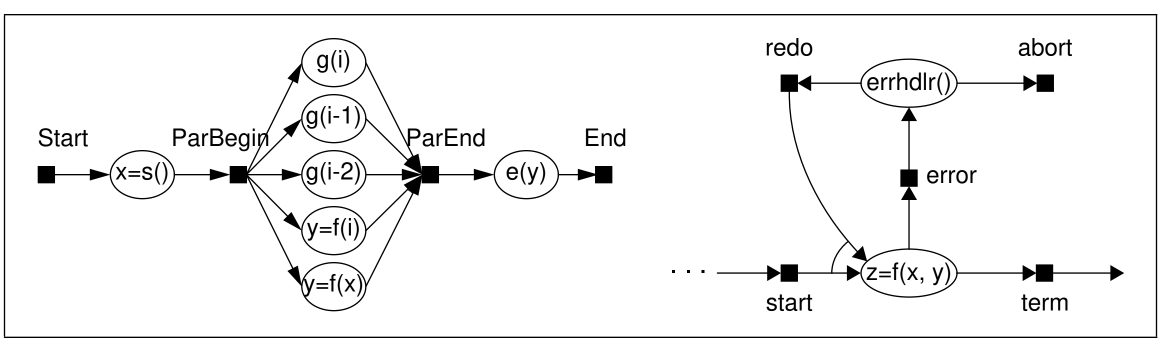

Figure 1: Pact execution plans.

of the action is successful now, the program can be continued — the term event is triggered. Note that the action's atomicity is responsible for this well-defined fault semantics which makes error-handling (and recovery) simple.

3 Pact's runtime system

The runtime system is designed to provide the features of the programming model in the most efficient way on a massively parallel distributed-memory computer. To do this, the following design decisions are used.

- The task of managing the execution and load-balancing, which is traditionally performed by a centralized scheduler, is done hierarchically to provide high scalability. The parallel execution is managed by a single process that also controls the data-flow in the program, while the computation-intensive dynamic load-balancing is done in a distributed way.
- Multithreaded servers are used to mask the latency of remote references to shared dataunits and scheduling of actions.
- Almost all message-passing in the runtime system is done asynchronously. While this is a general principle to obtain good performance on multitasking operating systems, it is also useful in this multithreaded server design.
- Virtual shared memory is implemented using a fixed distributed manager algorithm similar to the algorithm in [12] adapted to Pact's consistency model and the architecture of the runtime system.
- A commit protocol that includes logging on stable storage makes actions recoverable and hence the execution fault-tolerant.
- To avoid bottlenecks in logging, there are multiple Log Managers in the runtime system. However, the commit protocol is kept simple (and therefore efficient) by associating each action to one Log Manager during its execution.
- To get load information without remarkable performance degradation, it is piggy backed on other messages. Load information (such as run-queue length, cpu-usage, page-faults, etc.) is needed to optimize the parallel execution of Pact programs by using a dynamic data-dependent load-balancing algorithm.

3.1 The execution of actions

As illustrated in Figure 2, the runtime-system consists of a number of system processes (Execution Manager, Supervisors, and Log Managers) and a number of processes generated by a precompiler from the Pact application program. The *Main Program*, where the execution plan is defined and serial execution is done, sends its action calls to the *Execution Manager*. Each action-call includes an (optional) *data-description* that describes the shared dataunits the action will read and write. Since the Supervisor S_x manages all dataunits i in the hash-class $x=h(i)$, the hashfunction is applied to the data-description. Based on this information, the Execution Manager forwards the action to the *Supervisor,* which manages most of the dataunits in the data-description.

This Supervisor schedules the action-call to "the best" *Server* process, where a thread is created to execute the action. The Supervisor's scheduling decision is based on a load-balancing strategy that uses information about the server node's actual load factor and data-locality information. To provide high performance with virtual shared memory, it is essential to avoid data-migration as far as possible. Therefore, one of the most important load-balancing criteria is the current ownership and replication of the dataunits. To do this, the Supervisor compares the nodes' ownership information and the action's data-description and uses these in the load-balancing strategy.

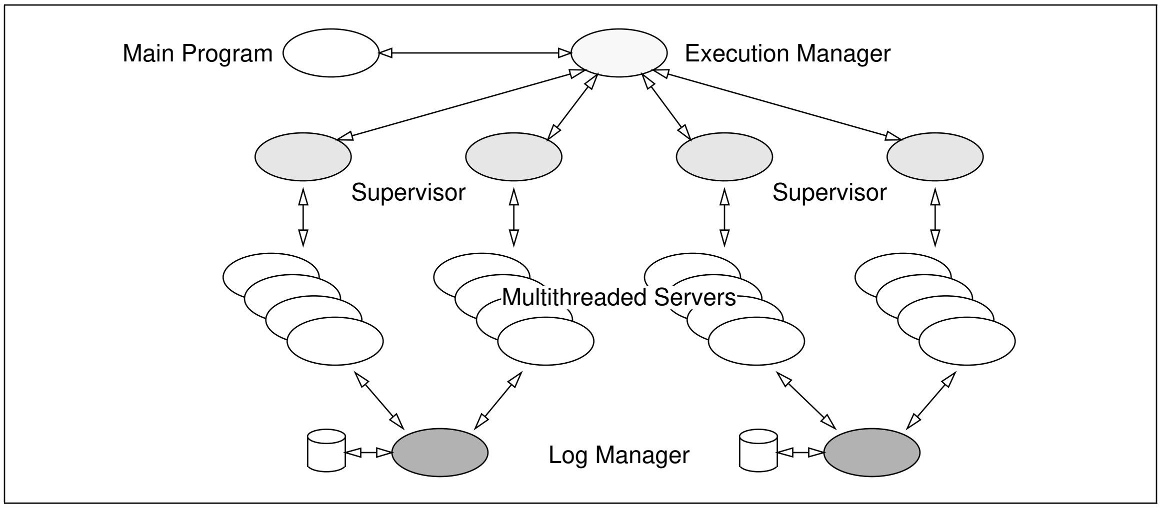

Figure 2: The Pact runtime system.

When the action needs access to a shared dataunit during its execution, it first looks in the directory of the server process. If its process is already the owner of the dataunit, it requests a lock for it. Otherwise, it sends a request to the Supervisor that manages this dataunit (and the next executable thread is executed). If the dataunit is not currently locked by the owner node, the Supervisor forwards the request to the owner node and changes the ownership information in its directory. After receiving the request, the owner node sends the dataunit to the requesting node, where the requesting thread is still waiting for it. At the end of the action's execution, it uses the following *commit* protocol, as shown in Figure 3.

1. The Server sends the action's action-identifier, its modified dataunits, and its result parameters to the Log Manager associated with the Supervisor from which the action-call is coming.
2. After the log data is written, the Log Manager sends the log sequence number (lsn) as an acknowledgment back to the Server.
3. The Server releases all local locks on shared dataunits and sends the action's results to the Supervisor.
4. The Supervisor releases the action's global locks.
5. The termination-event is triggered and the results are returned to the Execution Manager, which forwards them to the Main Program.

Pact's actions do not have the *durability* property of transactions; with distributed Log Managers (which do not give a global order of log records), it is necessary to write the log record to stable storage before releasing the locks on shared dataunits and returning the results to the Supervisor to enable recovery. To improve the performance of logging, a concept similar to *group commit* in [8] is used here. The idea of group commit is that the log buffer is not written before it is full, or before a specific time has passed.

When an error occurs inside of an action, the runtime system has to undo all the action's changes on shared data, due to the atomicity of actions. This is done by performing *abort* instead of commit. In the abort phase the action receives the previous (the last committed) state of the dataunits from the Log Managers, releases all locks and sends an error message to the Supervisor. The error-event is triggered, then, depending on the execution plan, an error handling action is scheduled, the error message is sent to the Main Program, or the action is just scheduled again, provided it was only a transient error that will not occur again.

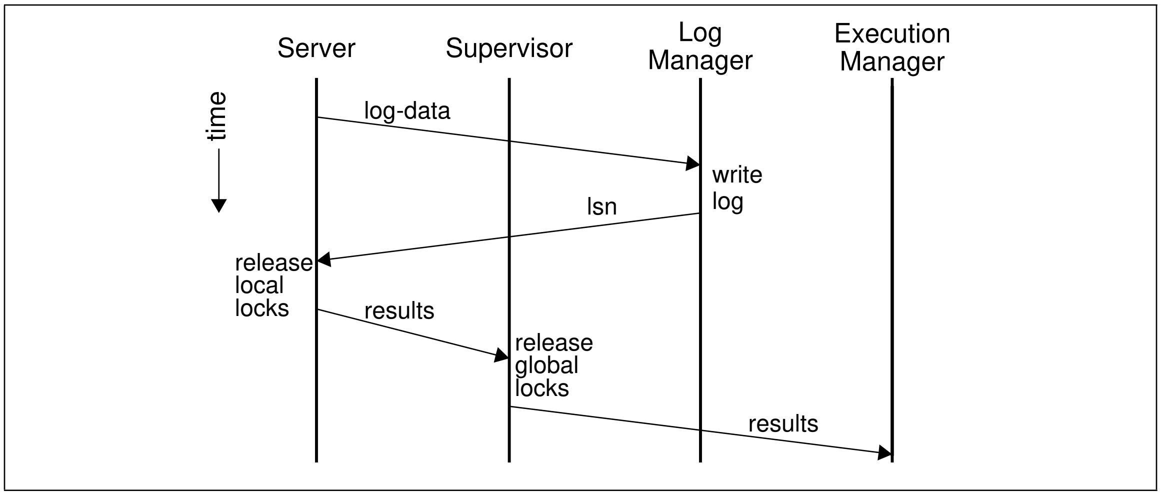

Figure 3: The commit protocol.

3.2 Recovery of partial failures

Fault tolerance in Pact is based on the *atomicity* of actions. This property gives actions simple fault semantics. Whenever a program error occurs or a server node fails, it is easy to put the shared data into consistent state by getting the lost dataunits in the last committed state from the Log Managers. Lost actions just have to be scheduled again.

A well-known problem for node failures is recognizing them. On Tandem systems this recognition is done by the operating system, based on I-am-alive messages. In systems that do not support this, the recognition must be done by the Pact runtime system itself. In order to do this, the Execution Manager periodically broadcasts are-you-alive messages to all nodes in the application. If it does not receive an answer during a time-out interval, it assumes that the node has failed. Although there is a slight chance that the decision is wrong, this does not cause serious problems; it is just necessary that all processes in the application ignore subsequent messages from this node.

The recovery algorithm for a server node works as follows:

1. After recognizing a node failure, the Supervisors mark all dataunits owned by the failed node as lost.
2. The Supervisors send all actions that have been running on the lost node to surviving server nodes.
3. Whenever an action requests lost dataunits, the Supervisor in charge forwards the request to all Log Managers in the system.
4. The Log Managers send the requested dataunit's log sequence number back to the Supervisor. The lsn includes a global timestamp[1] to enable a global ordering of the dataunit's log records.
5. The Supervisor decides on the basis of the lsn which is the last committed version and sends the request to this Log Manager.
6. The Log Manager sends the dataunit to the requesting node. The Supervisor marks the dataunit as owned by the requesting node.

Pact has the potential to guarantee the continuous execution of the parallel program by tolerating single server node failures; this presumes, however, that the underlying hardware and software tolerate partial failures as well. This presumption is correct for the Tandem machines on which

[1] The algorithm assumes the existence of a high-resolution global clock. If the system has no such clock, version numbers of dataunits can be used for ordering.

Pact's first prototype was implemented, but it does not apply to most other parallel computers. Often neither the operating system software nor the hardware is designed to tolerate partial failures. Therefore, the Pact runtime-system described in this article deviates from the design in [16], where the goal is to provide continuous execution. Here, it is necessary to improve the performance of restart recovery. To do this, it is necessary to accelerate the recovery of the Main Program and the Execution Manager by writing checkpoints on the log. Failures of the nodes where the Main Program, the Execution Manager, and the Supervisors are running are handled with the restart recovery algorithm by virtue of its simplicity.

3.3 Checkpoints

Checkpoints are written relatively infrequently because checkpoints are not required for recovery, but rather are an optimization to accelerate restart recovery. The Main Program and the Execution Manager write process-oriented checkpoints with the following protocol.

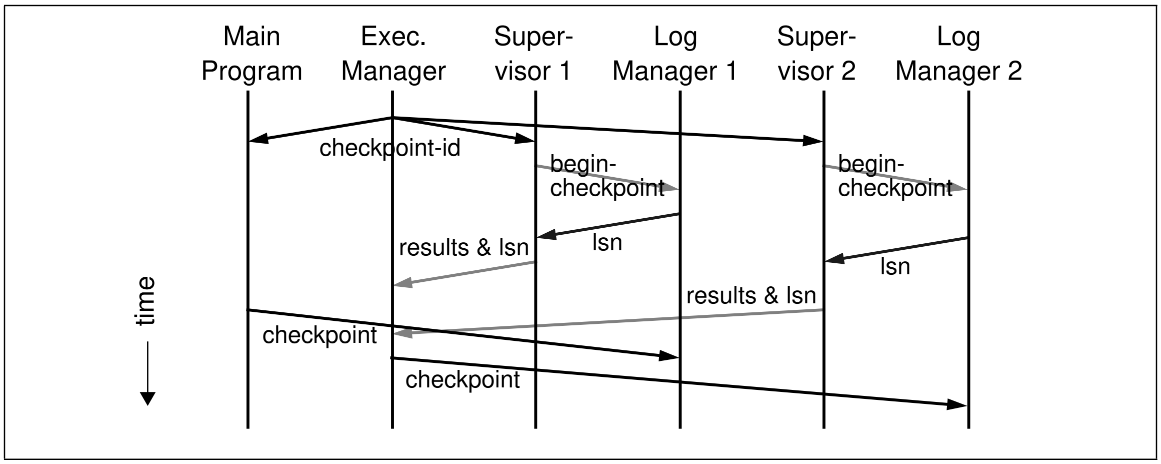

Figure 4: The asynchronous checkpointing protocol.

1. After a certain number of actions, the Execution Manager decides that it is time to checkpoint. It stops the forwarding of results to the Main Program and sends the checkpoint identifier to the Main Program and all Supervisors.
2. Each Supervisor sends a checkpoint record, which consists of the checkpoint identifier and the list of running actions (the Supervisor has not yet seen the result), to its Log Manager.
3. After the Supervisor receives the message with the log sequence number (lsn) of the checkpoint record from the Log Manager, it sends the results of all actions that have already committed, but whose results have not yet been forwarded, to the Execution Manager. This phase is finished by sending the checkpoint lsn to the Execution Manager.
4. The Main Program sends its checkpoint data together with the checkpoint identifier to the Log Manager. After doing this, it sends an acknowledge message back to its Execution Manager.
5. The Execution Manager sends its checkpoint data to its Log Manager.
6. The Supervisor, the Main Program, and the Execution Manager resume their execution.
7. The Log Manager computes a new *low-water mark* by comparing the list of incomplete actions of the actual Supervisor checkpoint CP_i with the list of the checkpoint CP_j to which the old low-water mark points. If $CP_i \cap CP_j = \emptyset$, then the low-water mark is set to CP_{j+1}. This comparison is done until the intersection is non empty or the low water mark is set to CP_{i-1}. Although this may sound like an expensive operation, in most cases only one comparison is needed (the low-

water mark points to CP_{i-2}); furthermore, the lists of incomplete actions are small and could be cached in main memory.

8. The checkpoint is completed by a checkpoint record of the Log Manager, which consists of the low-water marks of the log and the lsns of all records of this checkpoint.

This protocol ensures that the Supervisors and the Execution Manager have the same view of the execution because this is a requirement for the restart recovery algorithm. Since checkpointing does not affect the execution of actions on the server nodes and the Supervisors, the overhead of checkpointing is just a slow-down of the Log Managers and a delay in forwarding new action calls to the Supervisors. Besides accelerating restart recovery, there is a second advantage of checkpointing. By using checkpoints and low-water marks for both checkpoints and dataunits, it is not necessary to keep the log file of the whole execution. Older parts of the log file that contain only outdated versions of dataunits and checkpoints can be deleted. To handle situations where dataunits have not been used for a long time so that the actual version is on an old part of the log, the log is reorganized by *copy forwarding*, which is similar to the algorithm in the Camelot transaction manager [7].

3.4 Restart recovery

After the processes have been restarted and have read their last checkpoint from the log, they can reach the last consistent state by redo-recovery. Redo-recovery in this context means that lost actions' execution after the checkpoint is simulated. This is done by sending action-calls to the Execution Manager. The actions are not scheduled and executed again; instead, the result parameters of preceding executions are fetched from the log. The algorithm works as follows:

1. The Log Managers reopen their log files and scan them for the last consistent checkpoints and the last committed version of the dataunits. The dataunit information is cached to speed up future accesses, The checkpoints are sent to the Main Program and the Execution Manager.
2. The Main Program and the Execution Manager recover their current state from the checkpoint.
3. The Execution Manager asks all Log Managers for a list of recoverable actions.
4. Each Log Manager reads the last Supervisor checkpoint and scans the log between the checkpoint low-water mark (found in the Log Manager checkpoint) and the current checkpoint for commit records of actions in the list of incomplete actions of the current checkpoint. Each of these actions is put onto the list of recoverable actions together with all actions with commit records between the checkpoint and the end of the log. The list is sent to the Execution Manager.
5. The Execution Manager executes all incomplete actions in the following way: First it looks into the list of recoverable actions. If the action has committed before the crash, the Log Manager is asked for the results and does not forward the action-call to a Supervisor. Instead, the Log Manager answers the call by sending the results to the Execution Manager. If an action-call is not in the list of recoverable actions, the Execution Manager forwards the call to the Supervisor (as described in Section 3.1) where it is scheduled to a server process.
6. After that, the Main Program starts sending action-calls to the Execution Manager to be executed in the same way.
7. When an action needs access to a shared dataunit, it looks in the directory of the server process – without success. Then, it sends a request to the Supervisor, which knows if the dataunit is lost or already recovered. In the latter case, the Execution Manager forwards the request to the owner node; otherwise it recovers the dataunit as in Section 3.2.

4 Experience with the current prototype

The current prototype is running on a Tandem™ VLX *shared-nothing* multiprocessor system. The Tandem hardware and the Guardian™ distributed operating system tolerate partial failures, thus enabling Pact to guarantee continuous operation [16]. Since the Tandem VLX is designed for commercial applications and not for numerical computations, it has no floating-point coprocessors or vector units. Therefore, the absolute runtime of the measurements is not comparable to that of any supercomputer or high-performance workstation. The measurements of the additional time needed to enable fault-tolerance (such as commit protocols, logging) are shown in Table 1. The following applications are used: a gaussian elimination method to solve a dense 200×200 linear equation system, a multiplication of two 200×200 matrices, and a finite element analysis of a 1000 element object, where a conjugate gradient solver is used to solve the sparse equation system.

application	with logging	without logging	overhead
gauss (200×200)	916 sec	843 sec	8.65%
matmult (200×200)	1960 sec	1885 sec	3.98%
fem (1000 elem.)	6047 sec	5896 sec	2.57%

Table 1: Log overhead in the Pact prototype on Tandem VLX.

The reason for the overhead is not disk I/O, but the messages for logging. Differences between the design described in this article and the Tandem prototype arises from the fact that due to the absence of threads in the Guardian C30 operating system, the masking of latency is done by using multiple processes per node instead of multiple threads per server. This causes higher overhead and reduces the efficiency of the virtual shared memory implementation. Because Tandem VLX systems are limited to 16 nodes, a centralized scheduler design is used.

5 Current work and conclusions

The design of the runtime system described in this article is currently being implemented on an intel® Paragon™ supercomputer. The Paragon supports the implementation by providing a global clock and efficient asynchronous message passing by using message coprocessors. Even future versions of the Paragon, which will have multiprocessor nodes, will be used efficiently by the Pact runtime system. Multiple threads of a server process will run on different processors on the same node. Unfortunately, the Paragon, like most other parallel computers, does not tolerate partial failures. Therefore, fault tolerance on the Paragon means restart recovery of crashed programs.

The Pact prototype on the Tandem system shows low overhead for fault-tolerant execution of parallel programs. This overhead will be reduced even more with the Paragon implementation owing to its lower message passing latency and higher bandwidth. Threads with low switching times will be used to mask the latencies. User-defined dataunits minimize false-sharing, which makes weak consistency protocols unnecessary. Distributed schedulers control the execution by using system load information and information about the locality of shared data to balance the load and avoid data transfers thereby increasing the efficiency of virtual shared memory.

In contrast to classical checkpointing algorithms, only a small amount of data — the modified dataunits and the action's reference parameters, not the whole virtual address space of the processes — has to be written on stable storage. Therefore, even if commits in Pact are done more often than classical checkpoints. the overhead should be comparable or even lower, by virtue of fast restart recovery or even fault-tolerant execution.

Pact promises not only efficiency and low overhead, but also an easy-to-use programming model. With its consistency model, which is easy to understand and makes synchronization simple, it reduces the mistakes that can occur when using shared data in parallel programs. The ease of error-handling with atomic actions and the encapsulation of shared data accesses in Pact offer good program readability and reusability, which is extremely important for large (industrial) codes.

6 References

[1] Joel Bartlett. A NonStop Kernel. In *Proceedings of the 8th Symposium on Operating System Principles*. ACM SIGOPS, December 1981.

[2] A. Borg, J. Baumbach, and S. Glazer. A Message Passing System Supporting Fault-Tolerance. In *Proceedings of the Symposium on Operating System Principles*, pages 90–99, Atlanta, October 1983. ACM.

[3] John B. Carter, Alan L. Cox, Sandhya Dwarkas, Elmootazbellah N. Elnozahy, David B. Johnson, Pete Keleher, Steven Rodrigues, Weimin Yu, and Willy Zwaenepoel. Network Multicomputing Using Recoverable Distributed Shared Memory. In *Proceedings of the Compcon '93*, San Francisco, February 1993.

[4] Flaviu Cristian. Basic Concepts and Issues in Fault-Tolerant Distributed Systems. In A. Kershmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, Dagstuhl Castle, Germany, July 1991. Springer-Verlag.

[5] Flaviu Cristian and Farnam Jahanian. A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 12–20, Pisa, Italy, September 1991. IEEE Computer Society Press.

[6] Emootazbellah N. Elnozahy, David B. Johnson, and Willy Zwaenepol. The Performance of Consistent Checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47, October 1992.

[7] J. L. Eppinger, L. B. Mummert, and A. Z. Spector. Camelot and Avalon: A Distributed Transaction Facility. Morgan Kaufmann Publishers, 1991.

[8] D. Gawlick and D. Kinkade. Varieties of Concurrency Control in IMS/VS FastPath. *IEEE Database Engineering*, 8(2):3–10, 1985.

[9] Jim Gray. Notes on Database Operating Systems. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science*, pages 393–481. Springer Verlag, 1978.

[10] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1992.

[11] David B. Johnson and Willy Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In *Proceedings of Seventh ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, August 1988.

[12] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[13] Kai Li, Jeffrey F. Naughton, and James S. Plank. Checkpointing Multicomputer Applications. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 2–11, Pisa, Italy, September 1991. IEEE Computer Society Press.

[14] Barbara Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[15] Joachim Maier. Pact - a Fault Tolerant Parallel Programming Environment. In *Proceedings of the Second International Workshop Software for Multiprocessors & Supercomputers: Theory, Practice, Experience*, Moscow September 1994. Institute for System Programming of the Russian Academy of Sciences.

[16] Joachim Maier. Fault Tolerance Lessons Applied to Parallel Computing. In *Proceedings of the Compcon '93*, pages 244–252, San Francisco, February 1993.

[17] Paul Messina. The Concurrent Supercomputing Consortium: Year 1. *IEEE Parallel & Distributed Technology*, 1(1):9–16, February 1993.

[18] Michael L. Powell and David L. Presotto. Publishing: A Reliable Broadcast Communication Mechanism. In *Proceedings of the Symposium on Operating System Principles*, pages 100–109. ACM SIGOPS, October 1983.

[19] Bruce A. Rafnel. A Transaction Approach to Error Handling. *Hewlett-Packard Journal*, pages 71–77, June 1993.

[20] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems*, pages 204–226, August 1985.

[21] C.T. Wilkes and R.J. LeBlanc. Rationale for the Design of Aeolus: A system programming language for an action/object system. In *International Conference on Computer Languages*, pages 107–122, IEEE Computer Society Press, New York, 1986.