System-Level versus User-Defined Checkpointing

Luís Moura Silva João Gabriel Silva

Departamento Engenharia Informática
Universidade de Coimbra
POLO II - Vila Franca
P-3030 - Coimbra
PORTUGAL
Email: {luis,jgabriel}@dei.uc.pt

Abstract

Checkpointing and rollback recovery is a very effective technique to tolerate transient faults and preventive shutdowns. In the past, most of the checkpointing schemes published in the literature were supposed to be transparent to the application programmer and implemented at the operating-system level. In the recent years, there has been some work on higher-level forms of checkpointing. In this second approach, the user is responsible for the checkpoint placement and is required to specify the checkpoint contents.

In this paper, we compare the two approaches: system-level and user-defined checkpointing. We discuss the pros and cons of both approaches and we present an experimental study that was conducted on a commercial parallel machine.

1. Introduction

Checkpointing allows long-running programs to save state at regular intervals so that they may be restarted after interruptions without unduly retarding their progress. It is a feasible technique for tolerating transient failures and to avoid total loss of work. Several checkpointing schemes have been presented in the literature [1].

Basically, there are four main approaches to tolerate failures by using a checkpointing technique:

(i) leave it entirely to the application programmer;
(ii) provide the support for checkpointing inside the compiler [2][3];
(iii) the underlying operating system provides automatic recovery, releasing completely the programmer from consideration of faults. The programmer is only required to specify the interval between checkpoints [4];
(iv) the support for fault-tolerance is provided by a run-time library. In this approach, the checkpoint placement is not transparent to the programmer.

If we leave to the application programmer the task of dealing with failures it could be complex and cumbersome. In an experiment reported in [5] a distributed version of the *Scrabble* game was transformed in a fault-tolerant version, without any support from the operating system. The main lesson learned from that experience was that catering for failures at the application-level, although feasible, demands a huge effort from the application programmer: the fault-tolerance routines represented about 50% of the source code and took about 3 months to debug. The conclusion was that system or library support must be provided to make parallel applications resilient to failures.

The compiler-based checkpointing schemes represent a quite promising approach [2][3]. The placement of checkpoints is transparent to the programmer and the idea is to exploit the knowledge of the compiler to insert the checkpoint at the best place and to exclude some irrelevant areas of memory in order to reduce the size of the checkpoint file. Although it can be an effective technique it lacks portability since not all the compilers will include support for checkpointing. It will also be difficult to use in parallel/distributed applications that communicate through message-passing since the compiler cannot determine the state of the communication channels at the time of the checkpoint.

Another solution is to implement the checkpointing scheme at the operating system level. Most of the schemes presented in the literature are based on this approach of system-level checkpointing [1]. The idea is to provide automatic recovery of applications without any effort from the programmers. The most important advantage of this approach is quite valuable: transparency. It has some limitations, though: a checkpointing scheme that is implemented at the system-level knows nothing about the semantics of the application. What is relevant is just the exchanging of messages and the corresponding send/receive events. The application is seen as a “black-box” and the checkpointing scheme has no knowledge about its internal characteristics. As a result, system-level transparent checkpointing mechanisms typically take gross measures, such as logging all messages or backing up all the processes.

The final alternative is to provide support for checkpointing through a run-time library. This approach is not transparent to the user: the checkpoint contents and the places where checkpoints should be taken have to be defined by the application programmer. We call this approach as *user-defined checkpointing*.

These two last schemes (transparent system-level and user-defined checkpointing) have their advantages and drawbacks and there has been some discussion about whether fault-tolerance should be handled transparently by the operating system or should be provided on top of the operating system [6][7].

In this paper, we describe the pros and cons of both approaches. Section 2 will present a qualitative analysis between these two approaches. Section 3 refers the system-level checkpointing algorithm, while section 4 presents a user-defined checkpointing scheme. Section 5 presents the results of an experimental study that was taken on a commercial parallel machine. Section 6 describes the related work and, finally, section 7 concludes the paper.

2. System-Level versus User-Defined Checkpointing: a Qualitative Analysis

Transparent checkpointing is straightforward to use since it does not require any programming effort from the application developer. It is in fact a very commendable feature to have support from the operating system for automatic and effortless recoverable applications [8].

However, it was recognized by some users from the HPC community [9] that it is preferable to allow the application programmer to choose what and when to checkpoint. A user-defined high-level checkpointing facility would simplify the task of state saving, provide increased functionality, remove some of the restrictions imposed by the operating system and finally reduce the amount of data that should be included in a checkpoint.

Obviously, there are pros and cons in the use of each approach. We will try to quantify them and the list of metrics can be the following:

- **Programmer Effort**: in user-defined checkpointing the programmers will have to specify what data should be included in the checkpoint, and where checkpoints should be taken within the application code. This can be a simple task for a programmer that has complete knowledge about the application, but perhaps a more complicated operation for other kinds of users;
- **Portability:** another important aim is to provide portability of the checkpoint files and a portable checkpointing scheme. This attribute goes against the transparency. Transparent system-level checkpointing schemes are hardly portable while run-time libraries make this goal much easier to achieve [6];
- **Checkpoint Size:** the first schemes of transparent system-level checkpointing were based on a global snapshot of the processor address state, including all the dynamic data of the operating system. For instance, this approach has been followed in [10]. More recent schemes, like the ones presented in [11][12], among others, restrict the checkpoint data to a snapshot of the application context (data segment, stack segment and execution context). In any case, transparent checkpointing usually have to save larger amounts of memory and some of them unnecessarily, since they are not able to determine what is the critical state of the application. It is quite usual that applications make use of large amounts of temporary data that does not need to be saved: they can be easily recalculated or re-initialized during the recovery operation. Thus, if the programmer has the freedom to specify exactly which data should be saved in a checkpoint operation it would reduce considerably the size of the checkpoint, and consequently, reduce the performance overhead of the checkpoint operation [13];
- **Flexibility**: in some particular cases, some users may find convenient to perform a data-driven or iteration-based checkpoint, rather than a "blind" time-triggered checkpoint. This was acknowledged as a very important feature in [9]: for the sake of flexibility and functionality, checkpoint/restart mechanisms should be accessible from the application programmer. This sort of high-level checkpoints can thus be used for other purposes, rather than be restricted to fault-tolerance. For instance, high-level data dumps can also be used to perform job-swapping across different systems, for post-processing analysis or data-visualization;
- **Restartability of Checkpoints**: operating systems like UNIX provide a virtual and uniform memory layout in homogeneous machines, making easier the restarting of a process on a different processor. However, there are some state attributes that are kernel-dependent. They cannot be saved and carried across different processors in a sensible fashion [14]. To assure restartability of the checkpoints the application program should not make use of kernel-dependent attributes. User-defined checkpoints are more easily portable and can be restarted on different systems. Transparent checkpoints are usually restricted to homogeneous hosts.

We have been working in both approaches (system-level and user-defined checkpointing) and we have learned some lessons about the practical use of them. In the next sections we will describe a system-level checkpointing algorithm and a user-defined checkpointing scheme. Both schemes have been implemented in the same parallel system and we have conducted an experimental study to take some conclusions about the previous metrics.

3. System-level Checkpointing

In [15] we have presented an algorithm to implement a coordinated global checkpoint for distributed applications. That algorithm has some similarities with another one presented in [11]. For lack of space we do not present details about the algorithm. The interested reader is referred to any of those two papers.

4. User-Defined Checkpointing

In this approach the checkpoints are generated at a higher-level based on data-structures and variables that the developer defines as part of the checkpoint contents. This allows a significant optimization in the size of the checkpoints since we just checkpoint the application data that is really essential for recovering in case of failure, instead of saving all the address space.

High-level checkpoints can be migrated across heterogeneous machines. The recovery system knows the data types of the critical data structures and the checkpoints can be saved in the XDR format, thereby allowing the use of the checkpoint files in heterogeneous systems. This is an important advantage over "core-dump" like transparent checkpoints that are limited to homogeneous systems.

The application developers are also responsible for the placement of checkpoints. Care should be taken to place the checkpoint primitives into some particular points of the code that correspond to consistent global states of the application.

The basic idea is to exploit the semantics of the application in order to get more efficiency. However, the use of this approach should be minimally intrusive. The use of checkpointing should not require a re-design or a major change in the applications and should never decrease the potential for parallelism. In our case, only the insertion of

some additional library calls are required: usually one call to indicate a point of checkpoint, another for restarting, and a third one to specify the data that needs to be saved in stable storage. The application developer always knows what data is important, and it can regenerate everything else.

There is obviously a trade-off decidion between portabiliy and transparency. Some users would prefer automatic fault-tolerance while other users would consider portability and efficiency as more important issues than transparency and do not mind the effort of instrumenting their code with some checkpointing primitives.

The advantages of user-defined checkpointing are stressed out by the opinion of real users of parallel computing. According to the report produced by the SIO initiative [16] checkpointing is a common request, preferably allowing the application developer to choose what and when to checkpoint, and without degrading too much the overall performance of the application. That same report includes a detailed description of 18 real applications: some of them were already instrumented with a checkpoint facility that was implemented by the application programmers, though it would be essential to get some support for checkpoint-and-restart from the system or a run-time library.

We have implemented a user-defined checkpointing tool for the Parix operating system and it will be presented in the next sub-section.

4.1 The CHK-LIB

The `CHK-LIB` is a system library that runs on top of the Parix Operating System [17]. It works primarily as a communication library and provides support for checkpoint-and-restart. Any user that is not interested in the fault-tolerance facilities can use `CHK-LIB` as a normal communication library instead of using the Parix system interface. The programming interface of `CHK-LIB` was inspired in the MPI standard [18] to facilitate the porting of existing MPI programs to Parix.

The library provides a user-defined checkpointing scheme. The available fault-tolerance primitives are presented in Figure 1.

```
int    CHK_Pack_chkp(void *ptr,int size);
int    CHK_Restart(void);
int    CHK_Checkpoint(void);
```

Figure 1: Fault-Tolerant Primitives of the `CHK-LIB`.

4.1.1 Checkpoint Contents

This scheme does not need to save the system variables: the process stack, processor registers, or any other information related to the hardware or the operating system. The contents of the checkpoint are chosen by the application programmer by using the **`CHK_Pack_chkp()`** primitive to specify the critical data of the application. From our experience, it has not been very difficult to know what is the data that should be packed in a checkpoint, provided the programmer has enough knowledge about the application.

4.1.2 Checkpoint Placement

The **`CHK_Checkpoint()`** routine saves the critical data of the application in stable storage. The place where the checkpoints are placed should correspond to a consistent state of the application. Usually, the programmer can reuse some of the already existing synchronization points of the application that represent natural consistent global states, like barriers, end of the same iteration, collective operations and global tests of convergence. In these points the state of the process stack is useless. This strategy avoids the overhead of forcing a global consistent state. For some classes of applications this is not an hard task, specially for SPMD codes where it is quite easy to choose consistent states.

Assuming the placement of checkpoints is correct, the rest of the checkpointing protocol is quite simple. It follows a non-blocking 2-phase commit protocol. There is no need to be concerned with the state of the communication channels or possible messages in-transit, since the consistency is forced at the application level.

4.1.3 Recovery Operation

The recovery operation involves the roll back of all the processes. There is no need to restore the program counter and the processes' stack. The state of the program is entirely restored by loading the values of the data-structures from the checkpoint file. Processes have to restart the execution of the program from the beginning of the `main()` procedure, since the checkpoints do not include the program counter. To distinguish between a normal start and a recovery restart all the processes have to call the **`CHK_Restart()`** routine primitive before starting the computation cycle. It returns a boolean value: if `TRUE` it means the program is restarting from a previous checkpoint. The critical data of the application is loaded from the last checkpoint file. Otherwise, if it returns `FALSE` it corresponds to the normal beginning: all the initialization code should be executed before going to the computation cycle.

The interested reader is referred to [19] for more details about this library.

5. Implementation Results: A Quantitative Comparison

We have implemented both schemes in the Parix operating system and the `CHK-LIB`. All the results were taken in a Xplorer Parsytec machine with 8 transputers (T805). Each processor had 4 Mbytes of main memory. All the processors can read and write directly to the file-system of the host machine, that is a Sun Sparc 2 Workstation.

5.1 Application Benchmarks

To evaluate the checkpointing schemes we have used the following application benchmarks:

- **ISING**: This program simulates the behaviour of Spin-glasses. Each particle has a spin, and it can change its spin from time to time depending on the state of its direct 4 neighbours and the temperature of the system. Above a critical temperature the system is in complete disarray. Below this temperature the system has the tendency to establish clusters of particles with the same spin. Each element of the grid is represented by an integer, and we executed this application for several grid sizes.

- **SOR**: successive overrelaxation is an iterative method to solve Laplace's equation on a regular grid. The grid is partitioned into regions, each containing a band of rows of the global grid. Each region is assigned to a process. The update of the points in the grid is done by a Red/Black scheme. This requires two phases per iteration: one for black points and other for red points. During each iteration the slave processes have to exchange the boundaries of their data blocks with two other neighbours, and at the end of the iteration all the processes perform a global synchronization and evaluate a global test of convergence. Each element of the grid is represented in double precision, and we executed this application for several grid sizes.
- **ASP**: solves the All-pairs Shortest Paths problem i.e. it finds the length of the shortest path from any node i to any other node j in a given graph with N nodes by using Floyd's algorithm. The distances between the nodes of the graph are represented in a matrix and each slave computes part of the matrix. It is an iterative algorithm. In each iteration there is one of the slaves which has the pivot row. It broadcasts its value to all the other slaves. We will solve the problem with two graphs of 512 and 1024 nodes.
- **NBODY**: this program simulates the evolution of a system of bodies under the influence of gravitational forces. Every body is modelled as a point mass that exerts forces on all other bodies in the system and the algorithm calculates the forces in a three-dimensional dimension. This computation is the kernel of particle simulation codes to simulate the gravitational forces between galaxies. We ran this application for 4000 particles.
- **GAUSS**: solves a system of linear equations using the method of Gauss-elimination. The algorithm uses partial pivoting and distributes the columns of the input matrix among the processes in an interleaved way to avoid imbalance problems. At each iteration, one of the processes finds the pivot element and sends the pivot column to all the other processes. We will solve two systems of 512 and 1024 equations.
- **TSP**: solves the travelling salesman problem for a dense map of 16 cities, using a branch and bound algorithm. The jobs were divided through the possible combinations of the 3 first cities.
- **NQUEENS**: counts the number of solutions to the N-queens problem. The problem is distributed by several jobs assigning to each job a possible placement of the first two queens. We solved this algorithm with 13 queens.

These two last applications follow the Task-Farming paradigm. All the others are SPMD applications that require a geometric decomposition of the application data.

5.2 Performance of User-Defined (UDC) versus System-Level Checkpointing (SLC)

Hardly any experience with implementation of automatic and user-defined checkpointing has been reported in the literature. In this study, we have implemented system-level transparent and user-defined checkpointing in the same system and we used the same parallel machine and the same benchmarks to allow a direct comparison.

Table 1 compares the overhead per checkpointing of the two main schemes: `STC` and `UDC`. Both schemes use a non-blocking technique to perform the checkpoints concurrently with the application. The checkpoints are first saved to a *snapshot area* and are written to stable storage by an additional thread in a concurrent way with the computation.

The overhead of `UDC` was much lower than the corresponding system-level transparent checkpointing scheme (`SLC`). The third column shows the percentage reduction. The average decrease was around 58%. This means we can achieve at least half of the overhead by using user-defined checkpointing. The major contributions to that difference are the size of the checkpoints and the execution of the checkpointing algorithm.

Applications	Overhead SLC	Overhead UDC	Reduction (%)
ISING 1280	2.898	0.941	67.5 %
SOR 768	7.049	3.656	48.1 %
GAUSS 512	5.429	2.803	48.3 %
GAUSS 1024	10.478	8.327	20.5 %
ASP 512	3.247	0.834	74.3 %
ASP 1024	4.155	1.672	59.7 %
NBODY	0.985	0.115	88.3 %

Table 1: Overhead per checkpoint (in seconds) for the SPMD applications.

In Table 2 we present the overhead of the two approaches when using the two Task-Farming applications: TSP and NQUEENS. In this class of applications the user-defined checkpoints are taken in a centralized way, by the master process. All the other (slave) processes do not need to be checkpointed since they are stateless processes. The system-level transparent scheme takes a distributed coordinated checkpoint of all the processes since it is not able to exploit the characteristics of the applications.

	Overhead SLC	Overhead UDC	Reduction (%)
TSP	0.795	0	100 %
NQUEENS	0.976	0	100 %

Table 2: Overhead per checkpoint (in seconds) for the Task_Farming applications.

The overhead of both applications with the user-defined checkpointing tool (`UDC`) was literally zero. We have executed those schemes several times, but the execution time with checkpoints was also equal to the normal execution time. This result is due to three main reasons:
(i) in `UDC` it is only necessary to perform a centralized checkpoint of the master process: the slave processes do not need to be checkpointed; (ii) the size of the checkpoint is negligible; (iii) the checkpoint operation is taken by an additional thread at the master processor and this operation does not disturb the normal execution of the computation.

Table 3 presents a comparison in the checkpoint duration. The first column represents the checkpoint duration of the system-level checkpointing scheme, while the second column represents the user-defined checkpointing mechanism.

The average reduction in the checkpoint duration was 62%. This is the price to pay for executing a distributed checkpointing algorithm to assure a consistent global state of the application. In the `UDC` approach the set of all the checkpoints already correspond to a global consistent state.

The checkpoints were placed in global points of synchronization already existing in the applications. These points correspond to consistent global states of the program.

Applications	System-Level Checkpoint	User-Defined Checkpoint	Reduction (%)
ISING 1280	18.364	12.209	33.5 %
SOR 768	14.580	8.922	38.8 %
GAUSS 512	10.792	4.493	58.3 %
GAUSS 1024	22.604	16.152	28.5 %
ASP 512	8.015	2.618	67.3 %
ASP 1024	14.824	8.441	43.0 %
NBODY	11.918	0.878	92.6 %
TSP	45.694	0.129	99.7 %
NQUEENS	22.652	0.128	99.4 %

Table 3: Checkpoint duration in seconds (system-level versus user-defined checkpointing).

It is worth to mention the substantial reduction that is achieved with the two Task-Farming applications: 99%. This is the difference in taking a centralized and a distributed checkpointing.

5.3 Memory Overhead of User-Defined versus System-Level Checkpointing

In Table 4 we compare the size of the checkpoints in the two approaches. The size is presented in Kbytes. The Table only presents the SPMD applications.

Applications	System-Level Checkpoints	User-Defined Checkpoints	Size Reduction
ISING 256	3176	269	91.5 %
ISING 512	3962	1049	73.5 %
ISING 768	5268	2341	55.5 %
ISING 1024	7082	4145	41.4 %
ISING 1280	9408	6461	31.3 %
ISING 1536	12251	9289	24.1 %
ISING 1792	15601	12629	19.0 %
SOR 256	3409	540	84.1 %
SOR 512	4999	2104	57.9 %
SOR 768	7613	4692	38.3 %
SOR 1024	11251	8304	26.1 %
SOR 1280	15994	12940	19.1 %
GAUSS 512	5312	2052	61.3 %
GAUSS 1024	11806	8200	30.5 %
ASP 512	3990	1024	74.3 %
ASP 1024	7230	4096	43.3 %
NBODY 4000	3540	312	91.1 %

Table 4: Size of checkpoints for SPMD applications (system-level versus user-defined).

The average reduction was 50.7%. This means that in the overall, 50% of the application state was not relevant from the point of view of assuring a successful restart of the application. With user-defined checkpointing we just checkpoint the data that really needs to survive to failures. All the other data corresponds to temporary variables or can be easily (if not automatically) reconstructed during the recovery phase. The system-level transparent checkpointing scheme saves all the data that belongs to the address space of the process in a blind way. No optimization was performed.

Table 5 presents a similar comparison in the size of the checkpoints for the two Task-Farming applications. The size is presented in bytes. In this case, the average reduction was 99.9%, and the previous conclusion for user-defined checkpointing is even more distinguishing.

Applications	System-Level Checkpoints	User-Defined Checkpoints	Size Reduction
TSP 16	2953728	912	99.9 %
NQUEENS 13	2913984	536	99.9 %

Table 5: Size of checkpoints for Task-Farming applications (system-level versus user-defined)

6. Related Work

There are tens of papers and technical reports in the literature about system-level checkpointing algorithms [1]. This approach has some nice advantages, like transparency and automatic recovery. However, it may incur a higher performance penalty, it requires more space in stable storage and is restricted to homogeneous platforms.

In the past years there has been some work on high-level checkpointing schemes [6][13][21-27].

Juan Leon implemented a transparent checkpointing mechanism for PVM (Fail-Safe PVM) [20]. Despite the feasibility of that mechanism he considered a possible relaxation of the transparency as an interesting alternative. Latter on, he proposed an application-oriented toolkit for checkpointing in message-passing systems whose main goal is to provide portability [21]. The application programmer should be responsible for specifying which portions of the application state must be checkpointed. This information will reduce the checkpoint size. The placement of checkpoints should be specified by some programming hints and should also use some of the synchronization points in the application.

Jim Plank implemented a checkpointing tool for sequential UNIX applications - *libckpt* [13]. It can be used in a mode completely transparent to the programmer, but also supports the use of checkpointing primitives to specify the contents of checkpoint. It has been proved to be a very effective tool and we have used it for checkpointing UNIX processes. Another semi-transparent checkpointing tool was presented in [22], although this one is restricted to the Intel Paragon multicomputer.

Another proposal was presented in [23]. That paper presents an object-based DSM system called Dome. That system was implemented on top of PVM and provides a library of distributed objects for parallel programming. The main goals of Dome are portability, load balancing and fault-tolerance. In this line, they have presented a portable checkpointing facility based on a set of C++ methods that can be used to checkpoint some of the program data structures. Those authors proposed two alternative methods for checkpointing: one that requires the help of a preprocessor and a second one that is fully transparent to the application. The preprocessing method inserts extra C++ statements into the code to facilitate the placement of checkpoints. The programmer still has to use those methods but can place a checkpoint almost everywhere in the code. The preprocessor is responsible for instrumenting the program with code to

save and restore the stack and the program counter. In our case, it was not necessary to save the stack contents or the program counter.

A more powerful approach was proposed in [24]. That paper presented a preprocessor that enables machine-independent checkpoints by automatic generation of checkpointing and recovery code. Although that scheme would alleviate the task of the programmer it would be necessary some more widespread use of that preprocessor.

A very interesting checkpointing scheme was presented in [25]. That scheme was implemented in Charm++, a parallel object oriented programming language. Some techniques were presented that automatically generate checkpointing code, albeit the programmer still has the possibility to specify the checkpoint contents and may have some control about the checkpoint placement.

Another user-defined checkpointing tool was developed on behalf of the FTMPS project [26]. That tool provided four checkpointing library calls (`cp_init`, `cp_define`, `cp_store` and `cp_deinit`) and has several similarities with our scheme.

In [27] was presented another high-level checkpointing scheme that also includes support for reconfiguration of the application in the occurrence of a partial permanent processor failure.

Finally, a checkpointing tool, called *libft* , was developed in the AT&T Laboratories [6]. Checkpointing is initiated through a user inserted function call and the user is responsible for specifying the critical data of the application.

All these papers support the idea that portability and user-defined checkpointing can be, in some cases, an alternative to system-level checkpointing.

7. Conclusions

This paper presents a comparison between two main checkpointing approaches: system-level versus user-defined checkpointing. In the user-defined approach the programmer is responsible for identifying the application data that should be saved and the places in the application code where the checkpoint calls should be inserted. Despite the increased complexity for the application programmer, user-defined checkpointing presents some interesting advantages, namely:

- reduction in the time overhead: the size of checkpoints can be made smaller than a “core-dump” checkpoint. This would reduce the checkpoint I/O traffic;
- reduction in the space required in stable storage. In some cases, this optimization can be considerable;
- the use of checkpointing can be tuned to the specific needs of the programmer. She will be responsible by the frequency of checkpointing and will have the freedom to save the real application state in some particular points of the application;
- if the high-level checkpoints are saved in the XDR format they can be used in heterogeneous architectures, while system-level checkpoints can only be migrated between homogeneous machines;
- programmer-induced recovery provides more flexibility. For instance, checkpoint-recovery can also be used to tolerate software bugs as was proposed in [7];
- user-defined checkpointing can be seen as a multi-purpose technique: for fault-tolerance, playback debugging or coarse-grained job-swapping.

To conclude, we do not claim that system-level checkpointing is worse than user-defined checkpointing. What we tried to prove was that user-defined checkpointing has some interesting advantages and it can be used in some situations, provided the programmer is willing to instrument her code with some additional fault-tolerance primitives.

Acknowledgements

We would like to thank the anonymous referees for their thoughtful comments. This work was partially supported by the Portuguese Ministry of Science and Technology (MCT), the European Union through the R&D Unit 326/94 (CISUC) and the project PRAXIS XXI 2/2.1/TIT/1625/95.

References

[1] E.N.Elnozahy, D.B.Johnson, Y.M.Wang. “A Survey of Rollback-Recovery Protocols in Message Passing Systems”, Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon University, October 1996

[2] C.C.J.Li, W.K.Fuchs. “CATCH - Compiler Assisted Techniques for Checkpointing”, Proc. 20th Fault-Tolerant Computing Symposium, FTCS-20, pp. 74-81, 1990

[3] J.Long, W.K.Fuchs, J.A.Abraham. “Compiler-Assisted Static Checkpoint Insertion”, Proc. 22nd Fault-Tolerant Computing Symposium, FTCS-22, pp. 58-65, 1992

[4] G.Barigazzi, L.Strigini. “Application-Transparent Setting of Recovery Points”, Proc. 13th Int. Fault-Tolerant Computing Symposium, FTCS-13, pp. 48-55, 1983

[5] P.McGrath, B.Tangney. "Scrabble - A Distributed Application with an Emphasis on Continuity", Software Engineering Journal, pp. 160-164, May 1990

[6] Y.Huang, C.Kintala. “Software Implemented Fault-Tolerance: Technologies and Experience”, Proc. 23rd Fault-Tolerant Computing Symposium, FTCS-23, pp. 2-9, 1993

[7] Y.M.Wang, Y.Huang, K.P.Vo, P.Y.Chung, C.Kintala. “Checkpointing and Its Applications”, Proc. 25th Fault-Tolerant Computing Symposium, FTCS-25, pp. 22-31, July 1995

[8] R.E.Strom, S.A.Yemini, D.F.Bacon. “Towards Self-Recovering Operating Systems”, in Parallel Processing and Applications, Elsevier Science Publishers, pp. 475-483, 1988

[9] P.Messina, T.Sterling. “Report on the Workshop on System Software and Tools for High-Performance Computing Environments”, Pasadena, CA, April 1992

[10] Y.Tamir, C.H.Sequin. “Error Recovery in Multicomputers Using Global Checkpoints”, Proc. 13th Int. Conf. on Parallel Processing, pp. 32-41, 1984

[11] E.N.Elnozahy, D.B.Johnson, W.Zwaenepoel. “The Performance of Consistent Checkpointing”, Proc. 11th Symposium on Reliable Distributed Systems, pp. 39-47, 1992

[12] J.S.Plank. “Efficient Checkpointing on MIMD Architectures”, PhD Thesis, Department of Computer Science, Princeton University, June 1993

[13] J.S.Plank, M.Beck, G.Kingsley, K.Li. “libckpt: Transparent Checkpointing Under UNIX”, Conference Proceedings USENIX Winter 1995 Technical Conference, January 1995

[14] J.Smith, J.Ioannidis. “Notes on the Implementation of a Remote Fork Mechanism”, Technical Report Columbia University, 1989, available by ftp: `ftp.cs.columbia.edu`

[15] L.M.Silva, J.G.Silva. “Global Checkpointing for Distributed Programs”, Proc. 11th Symposium on Reliable Distributed Programs, Houston USA, pp. 155-162, October 1992

[16] Scalable I/O Initiative, Applications Working Group, Working Paper No. 1, available at: `http://www.ccsf.caltech.edu/SIO/SIO.html`

[17] “Parix 1.2: Software Documentation”, Parsytec Computer GmbH, March 1993

[18] MPI Forum. "A Message Passing Interface Standard", March 1994, available at: `http://www.netlib.org/mpi/`

[19] L.M.Silva, “Checkpointing Mechanisms for Scientific Parallel Applications”, PhD Thesis presented at the Univ. of Coimbra, Portugal, January 1997, ISBN 972-97189-0-3

[20] J.Leon, A.L.Fisher, P.Steenkiste. “Fail-Safe PVM: A Portable Package for Distributed Programming with Transparent Recovery”, Technical Report CMU-CS-93-124, Dep. Computer Science, Carnegie-Mellon University, February 1993

[21] J.Leon, “An Application-Oriented Toolkit for Highly Available Distributed Scientific Computing”, PhD Thesis Proposal, Department of Computer Science, Carnegie-Mellon University, 1994

[22] Y.Chen, J.Plank, K.Li: “CLIP: A Checkpointing Tool for Message-Passing Parallel Programs", Proceedings of Supercomputing’97, San Jose, California, November 1997

[23] J.Arabe, A.Beguelin, B.Lowekamp, E.Seligman, M.Starkey, P.Stephan. “Dome: Parallel Programming in a Heterogeneous Multi-User Environment”, Technical Report, Carnegie-Mellon University, April 1995

[24] B.Ramkumar, V.Strumpen. “Portable Checkpointing for Heterogeneous Architectures”, Proc. 27th Fault-Tolerant Computing Symposium, FTCS-27, Seattle, June 1997

[25] S.Krishnan, L.V.Kale. “Efficient, Machine-Independent Checkpoint and Restart for Parallel Programs”, Technical Report, Dep. Computer Science, University of Illinois, 1994

[26] G.Deconinck, J.Vounckx, R.Lauwereins, J.Peperstraete. “A User-Triggered Checkpointing Library for Computation-Intensive Applications”, Proc. of the 7th Int. Conf. on Parallel and Distributed Computing and Systems, Washington DC, pp. 321-324, Oct. 1995

[27] V.Naik, S.Midkiff, J.Moreira: "A Checkpointing Strategy for Scalable Recovery on Distributed Parallel Systems", Proceedings of Supercomputing’97, San Jose, California, November 1997