computing@computingonline.net
www.computingonline.net

Print ISSN 1727-6209
On-line ISSN 2312-5381
International Journal of Computing

ABSTRACTION CHECKPOINTING LEVELS: PROBLEMS AND SOLUTIONS

Bakhta Meroufel, Ghalem Belalem

Department of Computer Science, Faculty of Exact and Applied Sciences
University of Oran, Algeria
e-mail: bakhtasba@gmail.com, ghalem1dz@gmail.com

Abstract: A common approach to guarantee an acceptable level of fault tolerance in scientific computing is the checkpointing. In this strategy: when a task fails, it is allowed to be restarted from the recently checked pointed state rather than from the beginning, which reduces the system loss and ensures the reliability. Several systems use the checkpointing to ensure the fault tolerance such as HPC, distributed discrete event simulation and Clouds. The literature proposes several classifications of checkpointing techniques using different metrics and criteria. In this paper we focus on the classification based on abstraction level. In this classification the checkpointing is categorized into two principal types: application level and system level. Each of these levels has its advantages and suffers from many problems. The difference between our present paper and the others surveys proposed in the literature is that: in this paper we will study each level in details. We will also study and analyze some works that propose solutions to solve the problems and exceed the limits of each abstraction level.

Keywords: Checkpointing, abstraction level, system level, application level, compiler, transparency, portability.

1. INTRODUCTION

Checkpointing/rollback recovery strategy has been an attractive approach for providing fault tolerant to distributed applications [2]. Checkpoints are periodically saved on stable storage sever and the recovery from a processor failure is done by restoring the system to the last saved state [3]. So the system can avoid the total loss of computations in case of the failure. One of the popular systems that use the checkpointing to ensure the fault tolerance is the distributed discrete event simulation. In this type of environment: the simulated system is partitioned into a set of sub-systems that are simulated by a set of processes that communicate by sending/receiving time stamped messages [42]. The state of each process in distributed discrete event simulation must be saved regularly to ensure a correct rollback in case of failures and decrease the system loss. However, it is proved that the performance of this system is dominated by the efficiency of the used checkpointing strategy. Thus, it is important to analyze and know more about this fault tolerance technique [43].

The checkpointing strategies can be classified according to their synchronization type [2]. Coordinated and uncoordinated are two fundamental approaches for checkpointing and recovery. There is another popular classification based on the abstraction level in which the state of a process is saved. There are a large number of design choices of abstraction level [6]. To understand this, let's consider the typical system stack shown in Fig. 1.

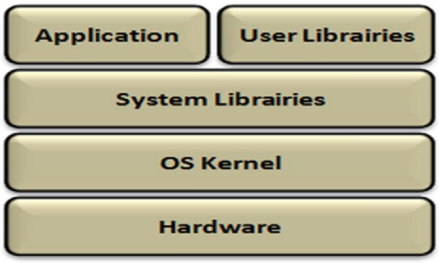

Fig. 1 – System stack.

It contains the original user application, which may be compiled/linked with user-level libraries. It may use system libraries, which resides on top of the OS kernel, which executes directly on top of hardware. Any of these levels may be modified with checkpointing functionality, so it possible to save and restore the stack levels above it. Furthermore, it is possible to insert new layers between these standard layers that enable checkpointing of the

layers above, such as the work proposed in [15] where Co-ordination Layer is created between the system library and the user library to ensure more transparency and to ensure also that system library will not be modified. The insertion of a new library can be necessary if the code of the existing library is note available or to ensure more transparency [15].

There are many survey papers proposed in the literature that study the abstraction level of checkpointing [1, 2, 5]. However, in our knowledge, none of them (survey) studied the solutions of the problems and limitations of each level. Our present work summarizes the majority of existing solutions of abstraction levels and proposes a comparative study between several papers.

This paper is organized as follows: in section 2, we propose a new abstraction level classification based on the transparency and we define in details each level. In section 3, we use some criteria to compare between the application level and the system level and we introduce some existing solutions for each problem caused by the abstraction level. We compare between the papers cited in our work in the fourth section. We finish our paper by a conclusion.

2. CHECKPOINTING LEVELS

There are many types of abstraction level classification in the literature. In [1], the classification is based on the implementation techniques. However, the authors in [2] and [5] propose other classification that uses the transparency as a criterion. In both previous works and in many other papers the application-level is referred as user-level. For our discussion we will distinguish them by their transparency with regard to the application program, further classified below. Our classification categories the checkpointing levels into three different types: application level, system level and mixed level (hybrid). Fig. 2 illustrates the proposed classification.

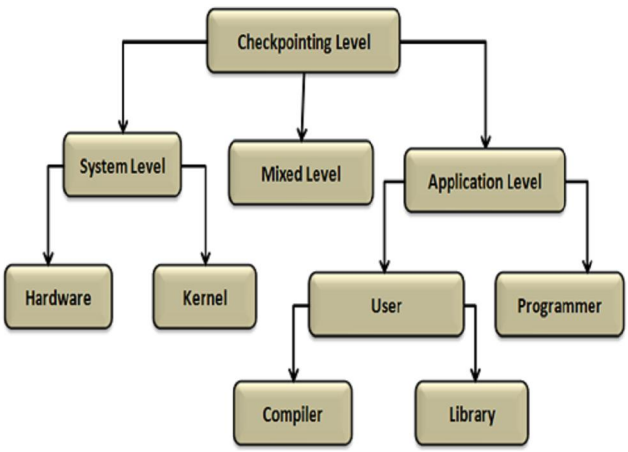

Fig. 2 – Checkpointing level classification.

2.1. SYSTEM-LEVEL (SLC)

System-level checkpointing is a technique which provides automatic, transparent checkpointing of applications at the operating system or middleware level. The application is seen as a black-box, and the checkpointing mechanism has no knowledge about any of its characteristics. Typically, this involves capturing the complete process image of the application. There are two main approaches of checkpointing at system-level: kernel (operating system) implementation and hardware implementation [2]. The system-level checkpointing can be activated by system call, Kernel-mode signal handler or Kernel thread.

2.1.1. KERNEL (OPERATING SYSTEM) LEVEL (SLC-K)

In kernel space every data structure relevant to a process's state is readily accessible: these include registers, memory regions, file descriptors, signal state, and more. This accessibility enormously simplifies the implementation of checkpoint/restart operations, though requires somewhat more knowledge of kernel internals. Berkeley Lab Checkpoint/Restart (BLCR) proposed in [17] is kernel level checkpointing in distributed system. It uses the coordinated checkpointing activated by a special thread named “Call back thread”. To reduce the overhead caused by the checkpointing, BLCR focuses on the management of I/O strategies. Transparent Incremental Checkpointer at Kernel level (TICK) [26] is another system that uses kernel level checkpointing. TICK considers the transparency as the most important criteria in scalable systems so the System-Level is the perfect Checkpointing to ensure transparency in grid calculations. TICK uses the buffered co-scheduling (BCS) [25] to ensure the checkpointing consistency. In BCS the messages are buffered and scheduled before transmission to omit the late and transit messages. ZAP [24] uses the kernel-level checkpointing for the migration. It provides a virtualization mechanism called Pod (Process domain) to cope with the resource consistency, resource conflicts, and resource dependencies that arise when migrating processes between machines with different persistent states. Stdchk proposed in [47] uses kernel-level checkpointing and it focuses on reducing the storage time by reducing the checkpointing size.

2.1.2. HARDWARE LEVEL (SLC-H)

Checkpointing may be supported by purpose designed hardware. As with operating system level implementations, this approach can be entirely transparent to users. But hardware-level

checkpointing is of limited importance precisely because it relies on custom hardware. The work proposed in [7] uses the checkpointing in the hardware level to tolerate faults in reconfigurable system. It assumes that each hardware module can be modeled by a Finite State Machine (FSM). This FSM will be extended to CFSM (checkpointed sFSM) by adding a new module related to FSM that control the checkpointing time (interval) and the placement of the checkpoint file. ReVive [22] requires modifications to the directory controllers of the machine that intercept the I/O to perform memory based distributed parity protection and logging in the background. The parity protection is used to protect the checkpoint file since it will be transferred via a network to others nodes to be stored. The logging is used to ensure the atomicity of transitions and omitting the modifications in memory in case of failure. In Logging buffering, the checkpoint value, is copied to a log, while the original location is modified and remains part of the working state. SafeNet [23] uses the same idea proposed in ReVive except that in SafeNet the parity protection is not used and the checkpoint files are stored only in the main memory. SafeNet uses uncoordinated checkpointing with pipelined validation rather than coordinated checkpointing used in ReVive. These differences justify why SafeNet requires more hardware resources than Revive.

2.2. APPLICATION-LEVEL (ALC)

A typical approach to avoid many of the complexities of checkpointing is based on taking an application centric point of view, and exploiting knowledge of the structure and behavior of a given application. In this approach the checkpointing is initiated, and to some degree managed, from within the application [5]. The application programmer identifies program points at which all essential state can be captured from within the application [21]. A common scheme of implementation is to install a signal handler for a default signal offered by the kernel to automatic-initiate the checkpoint operations. The signal handlers are defined at user-level and invoked by the kernel. This signal can be triggered by a timer that periodically interrupts the application [1]. The application level can be classified according to the transparency for the user. It means how much the user is involved in the process of checkpointing [2].

2.2.1. PROGRAMMER LEVEL (ALC-P)

It is called also manual code insertion. In this level the programmer manually inserts the checkpointing code in the application code in order to save its state and to recover after a fail-stop failure. The programmer inserts code at points in the application where he wants checkpointing to occur. The work introduced in [27] determines when the checkpoints are taken by identifying the main controlling loops in the benchmarks (usually the outer loops associated with major program phases), and inserts the checkpointing calls at the top of each loop iteration. The main advantage of this approach is that semantic information about memory contents is available when saving and recovering checkpoint data. Using this approach, only the important data necessary to recover the application are saved. The main drawback is that the programmer has to manually insert CPR (CheckPoint Recovery) code to save and recover an application state which is a very error prone process. Other drawback of this approach is the need to have access to the application source code.

2.2.2. USER LEVEL (ALC-U)

A user-level checkpointing is implemented in user-space and typically provides transparency by virtualizing all system calls into the kernel. Within this virtualized environment the checkpointing approach is able to capture the state of the entire process without being tied to the kernel and without modifying the application code, it just inserts the checkpointing call in the code using a library that will be activated at each execution or by using a special process named pre-compiler or compiler.

A. LIBRARY CHECKPOINTING:

This technique provides 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. Its implementation is based on the LD_PRELOAD environment variable [1] which installs the signal handlers and loads the checkpoint library without recompiling again the application. It can be implemented also by the signal handler. Fail-safe PVM (Parallel Virtual Machine) proposed in [8] implements a checkpointing library on top of Unix to support the fault tolerance (user-level). It uses the coordinated checkpointing activated by the daemon process to assure the coherence and it replicates the checkpoint files in many nodes to ensure their availability.

DejaVu [16] provides a user-level checkpointing by implementing a new library in the system. DejaVu is a coordinated checkpointing system, but unlike Distributed Snapshots it uses a novel runtime mechanism called OLP (On Ligne Protocol) to capture the state of communication channels as part

of the checkpoint and does not incur the overhead associated with flushing the network. The OLP is used to implement a loosely coordinated checkpointing. In contract to classic approaches of user-level checkpointing where the kernel state are recreating during the roll back of the system in case of failure, the paper [10] proposes to encapsulate the kernel state and to store it in order to recreating the same kernel state in the rollback phase, so the overhead will be decreased. The paper [10] proposes also a new system where the process father can control its sons and ensures the checkpointing service for them which reduces the size of checkpointing files.

The paper [11] describes a user-level checkpointing library to checkpoint multithreaded programs that use the POSIX threads library. It solves the problem of inter-blockage of processes that can be occurring in the checkpointing process. MTCP introduced in [12] and [13] focuses also on the user-level checkpointing in multithreading system and it uses coordinated checkpoints in shared memory system. DTMC proposed in [18] extends MTCP by focusing on the management of sockets and it proposes to use a single started thread in the rollback process to minimize the checkpoint size. In work [14], the user-level library checkpointing is used for the migration of threads. In order to reduce the time of migration, the paper proposes to synchronize between the source and the destination before the migration. It uses also the incremental checkpoint.

B. PRE-COMPILER CHECKPOINTING:

To overcome the problem of transparency to the user, the pre-compiler checkpointing approach is introduced. The basic idea for a program transformation tool or pre-compiler is to analyze the application source code and determine what program variables must be saved at each checkpoint. It also adds the appropriate code to the source code to write checkpoints and to restart the application from these checkpoints [28].

Compiler based approaches to checkpoint/restart fault tolerance are typically composed of two components: a pre-compiler, and a runtime support library. The pre-compiler is a source-to-source compiler that augments an existing application with calls into the associated runtime support library in order to provide transparent checkpoint/restart capabilities. This approach is independent of the MPI implementation. It permits us to implement the coordination protocol without modifying the underlying MPI library, which promotes modularity and eliminates the need for access to MPI library code which is proprietary on some systems. The additional requirement for the programmer is that he needs to insert calls to checkpointing functions at points in the application where he wants checkpointing to occur. The Fig. 3 shows the pre-compiler architecture.

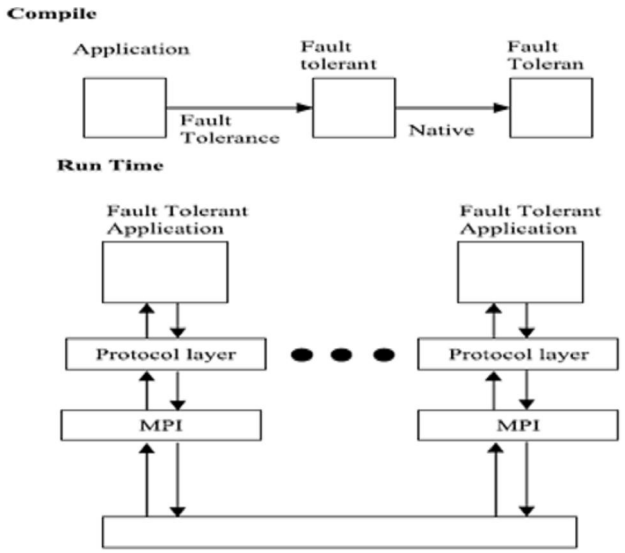

Fig. 3 – Pre-compiler architecture.

Many compilers are proposed in the literature such as in [29-34]. In [28], the authors propose a CPPC (Controller/Pre-compiler for Portable Checkpointing), it focuses on the automatic insertion of fault tolerance into long-running message-passing applications. It is designed to allow the execution restart on different architectures and/or operating systems. CPPC supports the checkpointing over heterogeneous systems, such as the Grid. It uses portable code and protocols, and generates portable checkpoint files while avoiding traditional solutions (such as process coordination or message-logging) by using safe points.

Safe point checkpoints are taken at the same relative code locations by all processes, without performing inter-process communications or runtime synchronization. To avoid problems caused by messages between processes, checkpoints must be inserted at points where it is guaranteed that there are no in-transit, nor orphan messages. Fig. 4 illustrates the unsafe zones in the communication between three processes P1, P2 and P3 with three local checkpoints $C_{i,j}$ for each.

Safe point identification and checkpoint insertion is automatically performed by the compiler CPPC. Among the systems that use the compiler we can cite also: [9, 15, 19 20]. The Distributed object migration environment (Dome) [9] addresses three major issues of distributed parallel programming: ease of use, load balancing, and fault tolerance and it uses SPMD system (Single Process Multi Data). The second and the third issues (load balancing, and fault

tolerance) are ensured by the checkpointing mechanism. The first issue (ease of use) in Dome is ensured by using a compiler that makes the checkpointing transparent to user. For the load balancing, Dome calculates the load of the processes based on execution speed and balances the load by redistributing the data in the system. This redistribution is assured by the compiler.

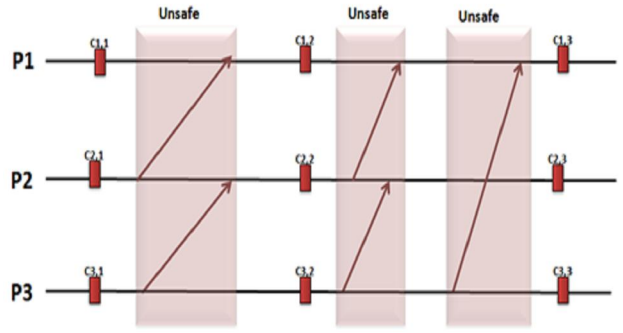

Fig. 4 – Safe/Unsafe points in three processes.

XCAT3 [20] uses the compiler in the user level checkpointing for grid computing. It focuses on the problem of checkpoint files availability so it proposes to use a federation of Storage services that is comprised of a Master Storage service and a set of Individual Storage services. XCAT3 exploits the blocking coordinated checkpointing to ensure the consistent global state. The work presented in [19] uses CPPC to implement the CPPC-G (CPPC for grid computing) on top of Globus4. CPPC-G extends CPPC by adding some others services to ensure the fault tolerance in the grid. The CPPC-G is charged to submit and monitor the CPPC applications. CPPC-G generates also the checkpoint files, detects the failures and automatically restarts the failed executions. C3 (Cornell Checkpoint (pre-Compiler) [15] exploits also ALC with a compiler to ensure more transparency for the user in scalable systems. According to [15], we cannot assume FIFO communication in message passing systems (MPI). In this case; C3 proposes a new strategy to detect consistent global states without FIFO assumption.

2.3. MIXED LEVEL CHECKPOINTING (MLC)

It is clear that neither application level checkpointing nor system level checkpointing is always an optimal solution to the system in term of performances. Efficiency and correctness are difficult issues for both approaches. So, Mixed-Level Checkpointing (MLC) combines aspects of both application and system level checkpointing. It can be used to develop checkpointing systems that are able to omit all the problems of each level type. The great difficulty in building MLC systems is separating the state of an application into logically consistent sets that can be checkpointed using either system or application level approaches. This approach was noted in [2] without any suggestion or implementation. However the paper [4] implements MLC as a system that balances between SLC and ALC according to the best for the performances. But it does not resolve the problem of the distinction between the states that must be stored with ALC or ALC. Since MLC in [4] balances between SLC and ALC so at any time the MLC can use only ALC so the programmer will modify the application code to support the checkpointing. In this case even the MLC is not transparent. For this reason our comparison is limited to System-level and Application-level.

3. SLC VERSUS ALC CHECKPOINTING: PROBLEMS AND SOLUTIONS

Obviously, there are problems and advantages in the use of each approach. We will try to quantify them using a list of metrics. And for each problem, we will indicate some solutions cited in the literature.

3.1. PROGRAMMER EFFORT (TRANSPARENCY)

This property refers to how the user perceives the fault tolerance solution. In application-level 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. So the SLC is more transparent then the ALC. Even in the ALC there are different transparency degrees for each type (ALC-P, ALC-U(Lib), ALC-U(Comp)). This can be illustrated in Fig. 5.

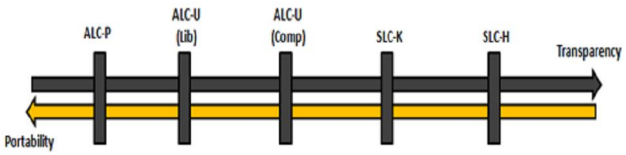

Fig. 5 – Transparency *vs* Portability in abstraction levels.

3.2. PORTABILITY

Another important aim is to provide portability of the checkpoint files and a portable checkpointing scheme. We say a checkpointing technique is portable if it allows the use of state files to recover the state of a failed process on a different machine. This attribute goes against the transparency (See Fig. 5). To solve the problem of portability in system level and even to improve it in application level,

some works use the virtual machines to ensure the consistency of processes states [8, 12, 17]. The second condition for a checkpointer to be portable is that all data is stored in a portable format so conversions may be made in case they are necessary for recovering the process state on a binary incompatible machine. Since different architectures represent data types in different sizes, a technique to convert data from one architecture to another meaningful data is needed. We can cite three strategies:

- Machine independent presentation technique: the classic example is XDR (External Data Representation) [36]. Some newer application-level schemes have used an XML-based format as well [37] or based on HDF5 (Hierarchical data forma 5) in CPPC [28].
- Lowest/Highest precision in the group [38]: means to save all checkpoint data in a precision that is higher/lower than that of any of the machines within the group. Both of these techniques suffer from the disadvantage that a conversion is required, even if the checkpoint is being restored to the machine on which it was taken. In addition, the technique of saving in the lowest precision of the group requires knowledge of the group members before checkpointing takes place.
- Receiver makes right technique [39]: the originator of the data simply checkpoints the data in its own precision. This technique has been used in [35]. In this case, the receiver is charged to ensure that the data conversion takes place when necessary. However, data conversion issues arise in the case of architecture differences.

3.3. CHECKPOINT SIZE

The size of full checkpoint file of SLC is larger than the ALC since the SLC is based on a global snapshot of the processor address state, including all the dynamic data of the operating system. In case the ALC, the programmer can precise exactly the needed data for the recovery [7, 17, 22, 23]. In [5] the authors prove that the ALC can reduce the checkpoints size by 50.7%.

However there are many techniques used to reduce this size of checkpoint file such as restricting the checkpointed data necessary for the rollback [14, 24]. The incremental checkpointing is also an efficacy strategy to decrease the checkpoint size. It consists to identify the dirty pages that have been modified since the last checkpoint, only the dirty pages are saved each checkpoint file. Many approaches use the incremental checkpoints [14, 16, 44, 47]. The authors in [26] propose different strategies of incremental checkpointing based on dirty pages such as Bookkeeping and Bookkeeping saving. The Word-level memory exclusion proposed in CAME [33] tracks every Read/Write operation so that both clean and dead memory can be excluded from checkpoint file, this leads to near optimally small files. However the overhead of word-level memory exclusion is too large.

3.4. 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. For the sake of flexibility and functionality, checkpoint/restart mechanisms should be accessible from the application programmer. This sort of ALC checkpoints can be used for other purposes rather than only the fault tolerance. For instance, ALS can also be used to perform job-swapping across different systems, for post-processing analysis or data-visualization. To increase the flexibility in system-level checkpointing many techniques of flexible checkpoints intervals are proposed in the literature [2]. To improve the flexibility in ALC-U, many potential checkpoints are placed in the code and among them (potential checkpoints) the checkpoint service can select the desired checkpoints according to the interval or the performances [29-32].

3.5. EFFICIENCY

The efficiency is evaluated by the overhead added to the application’s execution by the checkpointing system. In practice, there are two important sources of overhead. The first overhead type is incurred in order to maintain information about the execution of the application that would be used if a checkpoint were taken. This overhead, which we call the checkpoint-free overhead, is paid whether or not any checkpoints are taken during execution and should be kept as small as possible using techniques cited in section 3-3.

The used checkpointing protocols (coordinate, uncoordinated and communication induced) can infect also the checkpoint-free overhead. The technique that causes more overhead is the coordinated then the communication induced and then the uncoordinated. The last technique used to minimize the checkpoint-free overhead is using in case of external checkpoint responsible (the checkpoint initiator does not participate in the application) rather than an interne (the checkpoint initiator is one of process of the executed application) [26]. The second type of overhead is the cost of writing checkpoint data to stable storage

whenever a checkpoint is taken, which we call the checkpointing cost. This overhead is proportional to the size of the checkpoint data. ALC techniques tend to incur a checkpoint-free overhead, whereas the SLC techniques generally do not. The checkpointing cost can be reduced by reducing the amount of data in each checkpoint. It can be also reduced by controlling the I/O system [48]. In BLCR proposed in [17] the I/O manger collects the writes in a buffer in order to minimize the I/O calls, this strategy reduces the overhead cost by 12%. Storage type described in [22] infects the overhead cost: it groups schemes into three classes based on were and how the checkpoint storage is protected from errors.

The first type is Safe External Storage where the checkpoint is stored in external storage that is assumed to be safe. The second storage type is Safe Internal Storage where the checkpoint is stored in main memory or other internal storage and made safe through redundancy across the nodes. And the last storage type is Specialized Fault Class where the checkpoint storage is not protected with redundancy across nodes. However, the system is not expected to recover from faults that can damage that storage. It is clear that the first type of storage increases the overhead compared to the other two types. There is another used technique to decrease the checkpoint cost called "Copy on write" where the checkpointing write is executed in the background of the application execution (the checkpoint service elect a process to do the checkpointing write and the application continue the execution in the same time [8, 12, 34].

3.6. RESTART-ABILITY

Operating systems like UNIX provides 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 the restart-ability of the checkpointing, the application program should not make use of kernel dependent attributes. The restart-ability is related to the portability so ALC is more easily portable and can be restarted on different systems. SLC usually restricted to homogeneous hosts. To facilitate this restart-ability, the Zap [24] uses the domain file as a virtualization that ensures the portability of the process and its resources. In [10] the kernel state is encapsulated in list of references and information to ensure the restart-ability. For the restart-ability in case of migration, the work proposed in [14] proposes synchronization between the source and the destination of the thread; and both source/destination store the thread state in the virtual address.

3.7. FORCED CHECKPOINTING GENERATION

ALC cannot be generated forcefully, because in application level checkpointing, the process state can only be saved when checkpoint generation code is reached during execution whereas in SLC, it can be saved at any moment, since the state is obtained directly from the main memory by a separate thread or process. The technique of potential checkpoints insertion in ALC cannot make the forced checkpointing possible but it reduces the time between the desired time of forced checkpoints call and the execution of the real checkpointing.

3.8. CORRECTNESS

The correctness is the ability of a checkpointing system to ensure that the application produces a correct result. Of all of the properties that we have listed here, correctness is arguable the most difficult to ensure. To solve the problem of correctness in SLC and ALC, the Mixed checkpointing level MLC is proposed in [4] (section 2.3).

The paper [27] uses the correctness in application level to reduce the checkpoints size. This paper investigates definitions of program correctness that view correctness from the application's standpoint rather than the architecture's standpoint. Under application- level correctness, a program's execution is deemed correct as long as the result it produces is acceptable to the user. So in case of the soft computing (programs that produce inexact and/or approximate outputs) it is not necessary to store some states called Soft states because they will be modified at every re-execution without touching the correctness of the application.

4. COMPARISON BETWEEN ALC AND SLC

To resume all the characteristics of both ALC and SLC using all the points cited in the previous section, we present Table 1 that summarizes the differences between the application and the system level checkpointing using the criteria cited in section 3. (Y=Yes and N=No).

In Table 2 we compare between the different techniques used to implement the SLC (section 2.1) [1]. The used criteria are: Type, transparency, flexibility, Automatic initiation (ho is the responsible to initiate the checkpoint call) and if it is necessary to stop the application during the checkpointing.

Table 1. ALC Versus SLC.

Level	Transparency [4]	Portability [5]	CP Size [5]	Correctness [1]	Flexibility [2]	Forced CP [6]	Efficiency [2]	Restartability [5]	Scalability [3]
SLC	Y	N	High	N	N	Y	N	Difficult	N
ALC	N	Y	Low	N	Y	N	Y	Easy	Y

The system signal is a general purpose signal provided by the system but called only by the user to perform the checkpointing (automatic initiation) which minimizes the transparency and the flexibility. Kernel-mode signal handler is based on the signaling mechanism offered by the kernel, it is a new specific signal added to the kernel for the purpose of checkpointing. The advantage of Kernel mode is that the checkpoint is initiated by the system; but as the previous signal (system call), it is hard to control the checkpointing so the flexibility is not assured. Applications may be flexibly checkpointed by using a specific thread to the application's process called Kernel thread. Here a kernel thread is created to perform the checkpoint/restart activities so the flexibility is assured.

Since the kernel thread is also a different process and, especially in a multiprocessor system, it might run in parallel with the application that can change some data while the kernel thread is saving them. In this case a mechanism to stop the application is necessary in order to guarantee data consistency. The System Call and the Kernel Mode signal handler approaches have the advantages of being executed behind the process that has to be checkpointed. In this way the actual process address space is still the same of the process running in user mode. Unlike the system and kernel signal that required a modification in the code, the kernel thread is completely transparent.

Table 2. Comparison between SLC Implementation.

SLC Implementation	Signal System	Kernel System	Kernel Thread
Type	General System	Special System	Special Thread
Transparency	N	N	Y
Automatic initiation	User	System	User
Flexibility	N	N	Y
Stopping Application When Chekpointing	N	N	Y

We propose also a comparison between all the cited works in this paper using some criteria presented in section 3. The comparison is illustrated in Table 3 for the approaches using SLC and Table 4 for the approaches using ALC. The word CP represents the CheckPointing. Table 3 illustrates the used abstraction level for each approach, and if the transparency or portability is considered (section 3.1 & section 3.2). In case of BLCR for example, the portability is not considered but it uses the virtual machines-VM-to improve it. The other criteria are: The checkpoints size (section 3.3), the storage type (section 3.5), the used implementation (section 2.1), the checkpointing technique (section 3.5), the checkpointing responsible (section 3.5) and finally the system type. Table 4 presents the same criteria for ALC.

Table 3. System Level Checkpointing (SLC) Techniques

Paper	Level	Transpa-rency	Portability	CP size	Storage type	Implemen-tation	CP Type	CP respon-sible	System Type
SafeNet [23]	SLC-H	Yes	No	Full	Safe External	/	Uncoord	Internal	Shared Memory
ReVive [22]	SLC-H	Yes	No	Full	Safe internal	/	Coord	Internal	Shared Memory
[7]	SLC-H	Yes	No	Full	Safe internal	/	Coord (T-FIFO)	Internal	Reconfigu rable System
BLCR [17]	SLC-K	Yes	No (+VM)	Full+ compression	Safe external	Signal handler	Coord	External (Callback thread)	MPI
TICK [26]	SLC-K	Yes	No	Incremental	Safe external	Kernel thread	BCS	External	MPI
ZAP [24]	SLC-K	Yes	No	Full	Specialized Class	System Call	Coord	Internal	Shared Memory
Stdchk [47]	SLC-K	Yes	No	Incremental	Safe internal	System Call	Uncoord	External	Grid computing

Table 4. Application Level Checkpointing (ALC) Techniques

Paper	Level	Transpa-rency	Portability	CP size	Storage type	Implemen-tation	CP Type	CP responsible	System Type
[27]	ALC-P	No	Yes	Full (hard state)	Safe external	Signal handler	Safe points	Internal	Shared Memory
File-Safe [8]	ALC-U (Lib)	No	Yes (+VM)	Full	Safe Internal	API Interface	Coord	Internal(P) /External (Rollback)	MPI
DOME [9]	ALC-U (Comp)	No	Yes	Full	Safe Internal	API interface	Uncoord	Internal	SPMD
[10]	ALC-U (Lib)	No	Yes	Full	Safe external	Signal handler	Coord	Internal	Shared Memory
[11]	ALC-U (Lib)	No	Yes	Full	Safe External	Signal Handler	Coord	Internal	Shared memory
MTCP [12]	ALC-U (Lib)	No	Yes (+VM)	Full	Safe External	Signal handler	Coord	External (MTCP thread)	Shared memory
[13]	ALC-U (Lib)	No	Yes	Full	Safe External	Signal handler	Coord	External	Shared Memory
DMTCP [18]	ALC-U (Lib)	No	Yes	Full	Safe External	LD-PRELOAD	Coord	External	Shared Memory
XCAT3 [20]	ALC-U (Comp)	No	Yes (+XML)	Full	Safe External	Signal Handler	Coord	External	Grid computing
[14]	ALC-U (Lib)	No	Yes	Incremental	Safe External	Signal Handler	Coord	Internal	Shared Memory
C3[15]	ALC-U (Comp)	No	Yes	Full	Safe external	Signal Handler	Coord	Internal	MPI/Shared Memory
DejaVu [16]	ALC-U (Lib)	No	Yes	Incremental	Safe External	LD-PRELOAD	loosely coord	External	MPI
CPPC-G [19]	ALC-U (Comp)	No	Yes (+HDFS)	Full+ Compression	Safe external	Signal handler	Safe Points	Internal	Grid Computing
[44]	ALC-U (Lib)	No	Yes (VM)	Incremental	Safe external	/	Coord	Internal	HPC
[45]	ALC-U (Lib)	No	Yes (VM)	Incremental	Safe external	/	Coord	Internal	Cloud
Infiniband [46]	ALC-U (Lib)	No	Yes	Incremental	Safe external	/	Coord/ Uncoord	Internal	HPC

5. CONCLUSION

The classification based on abstraction level illustrates the state of process saving. It is categorized into two principal levels: application and system levels. In this paper we have presented each abstraction level in details and we have compared between the levels using different criteria cited in many papers in the literature. The transparency and the portability are two principal criteria of the comparison and each of them expresses the contraire of the other.

What is new in this paper is that we have summarized the existing solutions for the majority of level problems and limits. We presented also some comparative studies between different works existing in the literature. It is important to note that is clear that the checkpointing technique whatever its abstraction level is not sufficient to ensure a good fault tolerance in a distributed system. So many papers propose lately to combine between the checkpointing and the replication techniques [40, 41]. This hybrid strategy can reduce the fault tolerance overload and execution time of the application itself since the replication ensures the fault tolerance in real time and the checkpointing reduces what suppose to be re-executed in case of failures.

6. REFERENCES

[1] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa and S. Jiang, Current practice and direction forward in checkpoint/restart implementation for fault tolerance, in *Proceedings of the 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, USA, (April 3-8, 2005).

[2] S. Siva Sathya, K. Syam Babu, Survey of fault tolerant techniques for grid, *Computer Science Review*, (4) 2 (2010), pp. 101–120.

[3] R. Garg and A. Kumar Singh, Fault tolerance in grid computing: state of the art and open issues, *International Journal of Computer Science and Engineering Survey*, (2) 1 (2011), pp. 88–97.

[4] G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali and P. Stodghill, Recent advances in checkpoint/recovery systems, in *Proceedings of the Next Generation Systems Program Workshop (IPDPS 2006)*, Rhodes Island, Greece, (April 25-29, 2006).

[5] L. M. Silva, J. G. Silva, System-level versus user-defined checkpointing, in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, West Lafayette, Indiana, (October 20-23, 1998), pp. 68–74.

[6] V. Fontes, B. Schulze, M. Dutra and F. Porto, Checkpointing-based rollback recovery for parallel applications on the InteGrade Grid Middleware, in *Proceeding of the 2nd Workshop on Middleware for Grid Computing*, Toronto, Ontario, Canada, (October 18-22, 2004).

[7] D. Koch, C. Haubelt and J. Teich, Efficient hardware checkpointing, concepts, overhead analysis, and implementation, in *Proceedings of the 15th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2007)*, Monterey, CA, (February 18-20, 2007), pp. 188–196.

[8] J. Leon, A. L. Fisher, and P. Steenkiste, Fail-safe PVM: a portable package for distributed programming with transparent recovery, *Technical report in Carnegie Mellon University*, February 1993.

[9] J. N. C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, DOME: parallel programming in a distributed computing environment, in *Proceeding of the 10th International Parallel Processing Symposium (IPPS-96)*, Honolulu, Hawaii, (April 15-19, 1996), pp. 218–224.

[10] P. Tullmann, J. Lepreau, B. Ford, M. Hibler, User-level checkpointing through exportable kernel state, in *Proceeding of the International Workshop on Object Oriented Operating System*, Seattle, Washington (October 27-28, 1996).

[11] W. R. Dieter, J. E. Lumpp, A user-level checkpointing library for POSIX threads programs, in *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (FTCS'99)*, Madison, Wisconsin, (June 15-18, 1999), pp. 224–227.

[12] M. Rieker, J. Ansel, and G. Cooperman, Transparent user-level checkpointing for the native posix thread library for Linux, in *Proceeding of the PDPTA'2006*, Las Vegas, Nevada, USA, (June 26-29, 2006), pp. 492–498.

[13] W. R. Dieter, J. E. Lumpp, User-level checkpointing for Linux: threads programs, in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Boston, Massachusetts, USA, (June 25-30, 2001), pp. 81–92.

[14] H. Abdel-Shafi, E. Speight, and J. K. Bennett, Efficient user-level thread migration and checkpointing on Windows NT clusters, in *Proceedings of the 3rd USENIX Windows NT Symposium*, Seattle, Washington, (July 12-15, 1999).

[15] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, C3: A system for automating application-level checkpointing of MPI programs, in *Proceeding of the 16th International Workshop Languages and Compilers for Parallel Computing (LCPC 2003)*, College Station, TX, USA, (October 2-4, 2003), Lecture Notes in Computer Science, *Springer*, Vol. 2958, 2004, pp. 357–373.

[16] J. F. Ruscio, M. A. Heffner, S. Varadarajan, DejaVu: transparent user-level checkpointing, migration, and recovery for distributed systems, in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium, IPDPS'07*, Long Beach, California, USA, (March 26-30, 2007), pp. 1–10.

[17] P. H. Hargrove and J. C. Duell, Berkeley lab checkpoint/ restart (BLCR) for Linux clusters, in *Proceedings of SciDAC*, 2006, Denver, CO, (June 25-30, 2006).

[18] J. Ansel, K. Arya, and G. Cooperman, DMTCP: transparent checkpointing for cluster computations and the desktop, in *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, Rome, Italy, (May 25, 2009), pp. 1–12.

[19] G. Rodriguez, X. C. Pardo, M. J. Martin, P. Gonzalez, Performance evaluation of an application-level checkpointing solution on grids, *Future Generation Computer Systems*, (26) 7 (2010), pp. 1012–1023.

[20] S. Krishnan, D. Gannon, Checkpoint and restart for distributed components in Xcat3, in *Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, (November 8, 2004), pp. 281–288.

[21] J. P. Walters, V. Chaudhary, Application-level checkpointing techniques for parallel programs, in *Proceedings of the Third International Conference on Distributed Computing and Internet Technology (ICDCIT'06)*, Bhubaneswar, India, (December 20-23, 2006), pp. 221–234.

[22] M. Prvulovic, Z. Zhang, and J. Torrellas, ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors, in *Proceedings of 29th International Symposium on Computer Architecture (ISCA 2002)*, Anchorage, AK, USA (May 25-29, 2002), pp. 111–122.

[23] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, SafetyNet: improving the availability of shared memory multiprocessors with global checkpoint/recovery, in *Proceedings of 29th International Symposium on Computer Architecture (ISCA'2002)*, Anchorage, AK, USA, (May 25-29, 2002), pp. 123-134.

[24] S. Osman, D. Subhraveti, G. Su, and J. Nieh, The design and implementation of zap: a system for migrating computing environments, in *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, Massachusetts, USA, (December 9-11, 2002).

[25] F. Petrini, K. Davis and J. C. Sancho, System-level fault-tolerance in large-scale parallel machines with buffered coscheduling, in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Santa Fe, New Mexico, USA, (April 26-30, 2004).

[26] R. Gioiosa, J.C. Sancho, S. Jiang and F. Petrini, Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers, in *Proceedings of the ACM/IEEE SC2005 Conference on High Performance Networking and Computing*, Seattle, WA, USA, (November 12-18, 2005).

[27] X. Li, D. Yeung, Exploiting application-level correctness for low-cost fault tolerance, *Journal of Instruction-Level Parallelism*, (10), (2008), pp. 1–18.

[28] G. Rodriguez, M. Martin, P. Gonzalez, J. Tourio, R. Doallo, CPPC: a compiler-assisted tool for portable checkpointing of message-passing applications, *Journal Concurrency and Computation: Practice and Experience*, (22) 6 (2010), pp. 749–766.

[29] C. Li, E. Stewart, W. Fuchs, Compiler-assisted full checkpointing, *Journal Software-Practice and Experience*, (24) 10 (1994), pp. 871–886.

[30] J. Long, W. K. Fuchs and J. A. Abraham, Compiler-assisted static checkpoint insertion, in *Proceedings of the Twenty-Second Annual International Symposium on Fault-Tolerant Computing (FTCS-22)*, Boston, Massachusetts, USA, (July 8-10, 1992), pp. 58-65.

[31] G. Rodriguez, M. J. Martin, P. Gonzalez, J. Tourino, A heuristic approach for the automatic insertion of checkpoints in message-passing codes, *Journal of Universal Computer Science*, (15) 14 (2009), pp. 2894–2911.

[32] A. N. Norman, S.-E. Choi and C. Lin, Compiler-generated staggered checkpointing, in *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR'04)*, Houston, Texas, (October 21-23, 2004), pp. 1-8.

[33] J. Plank, M. Beck, G. Kingsley, Compiler-assisted memory exclusion for fast checkpointing, *IEEE Technical Committee on Operating Systems and Application Environments*, (7) 4 (1995), pp. 10–14.

[34] G. Bronevetsky, D. Marques, K. Pingali, S. A. MacKee and R. Rugina, Compiler-enhanced incremental checkpointing for OpenMP applications, in *Proceedings of the 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009)*, Rome, Italy, (May 23-29, 2009), pp. 1–12.

[35] H. Jiang, V. Chaudhary, J. Walters, Data conversion for process/thread migration and checkpointing, in *Proceedings of the 32nd International Conference on Parallel Processing (ICPP 2003)*, Kaohsiung, Taiwan, (October 6-9, 2003).

[36] B. Lyon, Sun external data representation specification, *Technical report RFC-1832, SUN Microsystems, Inc., Mountain View*, 1984.

[37] S. Krishnan, D. Gannon, Checkpoint and restart for distributed components in XCAT3, in *Proceedings of the 5th International Workshop on Grid Computing (GRID'2004)*, Pittsburgh, PA, USA, (November 8, 2004), pp. 281-288.

[38] B. Ramkumar, V. Strumpen, Portable checkpointing for heterogeneous architectures, in *Proceedings of the Twenty Seventh Annual International Symposium on Fault-Tolerant Computing, (FTCS-27)*, Seattle, Washington, USA, (June 24-27, 1997), pp. 58-67.

[39] H. Zhou, A. Geist, Receiver makes right data conversion in PVM, in *Proceedings of the IEEE Fourteenth Annual International Phoenix Conference on Computers and Communications*, Scottsdale, Arizona, USA, (March 28-31, 1995), pp. 458-464.

[40] D. Sun, G. Chang, C. Miao, X. Wang, Analyzing, modeling and evaluating dynamic adaptive fault tolerance strategies in cloud computing environments, *The Journal of Supercomputing*, (66) 1 (2013), pp. 193–228.

[41] U. Song, J. Gil, S. Hong, Checkpoint sharing-based replication scheme in desktop grid computing, in *Proceedings of the International Conference on Embedded and Multimedia Computing Technology and Service*, Gwangju, Korea, (September 6-8, 2012), *Lecture Notes in Electrical Engineering*, Vol. 181, 2012, pp. 477–484.

[42] Y.-B. Lin, Design issues for optimistic distributed discrete event simulation, *Journal of Parallel and Distributed Computing*, (62) 3 (2002), pp. 327–335.

[43] L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, Incremental checkpointing with application to distributed discrete event simulation, in *Proceedings of the Winter Simulation Conference (WSC 2006)*, Monterey, California, USA, (December 3-6, 2006), pp. 1004–1011.

[44] K. B. Ferreira, Rolf Riesen, Patrick Bridges, Dorian Arnold, Ron Brightwell, Accelerating incremental checkpointing for extreme-scale computing, *Journal of Future Generation Computer Systems*, (30) 1 (2014), pp. 66-77.

[45] H. Li, L. Pang, Z. Wang, Two-level incremental checkpoint recovery scheme for reducing system total overheads, *PLoS ONE*, (9) 8 (2014), Article ID e104591.

[46] K. Sato, A. Moody, K. Mohror, T. Gamblin, B. R. de Supinski, N. Maruyama, and S. Matsuoka, A User-level infiniband-based file system and checkpoint strategy for burst buffers, in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014)*, Chivago, IL, (26-29 May 2014), pp. 21-30.

[47] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, A. Gharaibeh, stdchk: a checkpoint storage system for desktop grid computing, in *Proceedings of the 28th International Conference on Distributed Computing Systems*, 2008, pp. 613-624.

[48] Z. Wang, X. Shi, H. Jin, S. Wu, Y. Chen, Iteration based collective I/O strategy for parallel I/O systems, in *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2014, pp. 287-294.

Bakhta Meroufel *is a PhD candidate in the Department of computer Science in the Faculty of exact and applied sciences at the University of Oran in Algeria.*

She received her M.S. degree in 2011 from the University of Oran, Algeria.

Her research interests are: distributed system, grid computing, cloud computing, fault tolerance, replication strategies and multi-agents systems.

Ghalem Belalem*: Graduated from department of computer science, Faculty of Sciences, University of Oran, Algeria, where he received PhD degree in computer science in 2007. He is now a research fellow of management of replicas in data replicas in data grid.*

His current research interests are distributed system; grid computing, could computing and data grid placement of replicas, consistency, fault tolerance, economic models, energy, Big data, and improved performance in large scale systems and mobile environment.