Discontinuous Incremental: A new approach towards extremely lightweight checkpoints

Viet Hai Ha and Éric Renault
Institut Télécom - Télécom SudParis
Samovar UMR INT-CNRS 5157
Évry, France
Email: viet_hai.ha@it-sudparis.eu and eric.renault@it-sudparis.eu

Abstract—**Checkpointing is an important method for providing fault tolerance, load balancing, process migration, periodic backup, and many other functions. It is also the basic tool used in CAPE, a paradigm which aims at distributing the execution of a program on a distributed-memory environment. This paper presents a new approach to checkpoint and an original optimization on the checkpoint structure that we have implemented and evaluated to make incremental checkpointing more efficient and more appropriate, especially for CAPE.**

I. Introduction

As a critical component to ensure a program will terminate its execution even in case of system or hardware failure, many works have been done to develop checkpointing techniques and different approaches have been used. The first solution referred to as complete checkpointing consists in saving all the information related to the running process [1]–[4], regardless these information have been modified from the beginning of the execution. This is not efficient for the text segment or when data can be retrieved easily for example from a dynamic shared library. The alternative to complete checkpointing is incremental checkpointing. In this case, only the information that have been modified since the beginning of the execution are effectively considered [5], [6]. The identification of the memory areas that have to be saved may either be provided by the developer of the application using e.g. directives and/or dedicated functions, or automatically be detected by the checkpointing tool itself. Both complete and incremental solutions have advantages and drawbacks. The main advantage of complete checkpointing is the simplicity of generation, while the main advantage of incremental checkpointing is the size of checkpoints. An important drawback of incremental checkpointing, as a consequence of the regular monitoring of the process memory, is that the execution speed may decrease significantly. Overcoming this disadvantage is an important issue to increase the performance of incremental checkpointer.

OpenMP (for Open Multi-Processing) [7] is a very simple and powerful set of directives and functions to generate parallel programs from C, C++ or Fortran codes. The main limitation of OpenMP is that it is limited to shared-memory architectures. Some attempts have been tried to port OpenMP on distributed memory architectures with various success. In order to achieve this goal, we have developed a new parallel computing paradigm called CAPE which aims at using checkpoints to distribute the execution of a program on a distributed-memory environment [8]–[10]. However, both complete and incremental checkpointing techniques cannot be used for an effective implementation. In the case of complete checkpointing, that was used in the first version of CAPE, the very large generated checkpoints significantly decreased the global performance. Althought incremental checkpointing can give smaller checkpoints, most current implementations do not have enough services to implement this paradigm efficiently.

As a result, in order to cope with both above requirements, this article presents a new approach to checkpointing and original optimizations on checkpoint structure that we have developed and evaluated with very good results.

The article is organized as follows: after the related works, Sec. III details our approach to generate discontinuous incremental checkpoints and also provides an evaluation of this approach. Sec. IV develops the data structure in checkpoint files, discussing memory granularity, the different storage format and arithmetics on checkpoints.

II. Related works

A. Incremental checkpointing

An incremental checkpoint consists in a file that stores the parts of the memory[3] that have been updated since the beginning of the execution of the program or the last stored checkpoint. From a high-level point of view, this is performed by setting access rights to read-only to all memory pages in order to force the system to deliver a SIGSEGV signal the next time a page is accessed for writing. Upon reception of the SIGSEGV signal, a copy of the content of the page is stored and access right to the memory page are restored to its initial values. When a checkpoint is required, the content of all modified page are compared to their initial content and the difference is stored in the checkpoint file.

In order to perform these operations, each process that may be checkpointed has to be associated a monitor. Typically, this monitor is in charge of 1) starting the process which checkpoints will be computed, 2) catching the SIGSEGV signals, 3) generating the checkpoint files and 4) waiting for

[3]References to memory are, unless otherwise specified, references to the virtual memory and not the physical memory. In the same way, the paper refers to virtual pages and virtual address spaces and not physical pages and physical address spaces respectively.

the termination of the process. However, this is not mandatory as a monitor may be attached to a process that is already-running.

Due to the execution of the monitor, the execution speed of the monitored process slightly decreases. Even when writing a single byte to a read-only memory region, a set of operations both in the kernel and in the monitor is executed: the kernel issues a SIGSEGV signal; the monitor catches the SIGSEGV signal, sets the access rights of the corresponding page to read/write after having read and saved its initial values. These operations increase the total time to finish the initial writing operation. If parts of the program contain a series of such operations, for example in case of initializing a large memory area from a constant or from values read in a database, this increase may become very large. Column 5, row 3 of Table II on page 4 highlights a case where the execution time is increased to nearly 10 times. The main issue remains on the way to avoid this decrease of speed while ensuring the recovery later.

B. CAPE

CAPE stands for Checkpointing Aided Parallel Execution. It consists in modifying a sequential program so that instead of executing each part the one after the other on a single machine, parts are distributed over a set of machines to be executed in parallel. CAPE is not intended to automatically detect which parts of the original code have to be executed in parallel. Such parts are identified by programmers while using OpenMP `pragma` directives. CAPE only aims at transforming a program so that it can be executed in parallel. A typical example is the distribution of a `for` loop. If the different iterations of a `for` loop are satisfying the Bernstein's conditions, i.e. any modified memory location is used only in the iteration loop where it is modified, it becomes possible to execute each loop iteration independently on different machines, compute the list of memory locations that have been modified in each iteration loop and include all these modifications inside a single process that will behave as if all iteration loops had been executed locally.

While using ckpt [4], a complete checkpointer, CAPE was proved its feasibility. Fig. 1 presents the general template for a `for` loops of the form `for ( A; B; C ) D;` when using the C programming language. In this template, functions `create ( before`$_i$ `)` and `diff ( before`$_i$`, after`$_i$`, delta`$_i$ `)` aim at extracting `delta`$_i$, the memory areas that have been updated on `host`$_x$, after this host has finished executing its part of the loop. The use of a complete checkpointer requires extra time for rendering this value and using an incremental checkpointer can avoid this. Another extra time is caused by functions `merge ( target, delta )` and `restart ( target )` which serve to inject these `delta`s into the memory of the initial host. The ability to directly execute this activity and to allow the process to resume its execution can avoid this extra time. This leads to some requirements like the ability to take and inject discontinuous incremental checkpoints in programs.

```
parent = create ( original )
if ( ! parent )
    exit
copy ( original, target )
for ( A ; B ; C )
    parent = create ( before$_i$ )
    if ( parent )
        ssh host$_x$ restart ( before$_i$ )
    else
        D
        parent = create ( after$_i$ )
        if ( ! parent )
            exit
        diff ( before$_i$, after$_i$, delta$_i$ )
        merge ( target, delta$_i$ )
        exit
parent = create ( final )
if ( parent )
    diff ( original, final, delta )
    wait_for ( target )
    merge ( target, delta )
    restart ( target )
```

Fig. 1. Template of CAPE for for loops.

III. Discontinuous incremental checkpointing implementation

A. Principle

Based on the incremental checkpointing approach, the main idea for discontinuous incremental checkpointing is to add information to indicate which sections should be checkpointed by the monitor. This information is provided using `pragma` directives and may be implemented using different mechanisms like signals, breakpoints, etc. Three `pragma` directives have been defined:

- `pragma dickpt start`
- `pragma dickpt stop`
- `pragma dickpt save` *filename*

Their behavior is described in Table I. In this table, "pages" refer to the pages of the virtual address space of the monitored process.

Assume a program is composed of four consecutive of segments (S1, S2, S3, S4) in which, only S2 and S4 need to be checkpointed and two checkpoints are taken in S2 and one in S4. Fig. 2 presents the prototype of the modified program, i.e including the directives that should be inserted to comply the above requirements.

B. Evaluation of performance

In order to highlight the impact of our new approach, we have developed a new checkpointer and measured its impact on a program computing the successive elements of a Markov Chain, see Fig. 3. In this figure, lines from 1 to 4 serve to include some C libraries. These inclusions do not effect the measured results. Two cases have been tested. The first one includes a directive to begin checkpointing at location 0 (line 10) and takes a checkpoint at location 1 (line 23), while

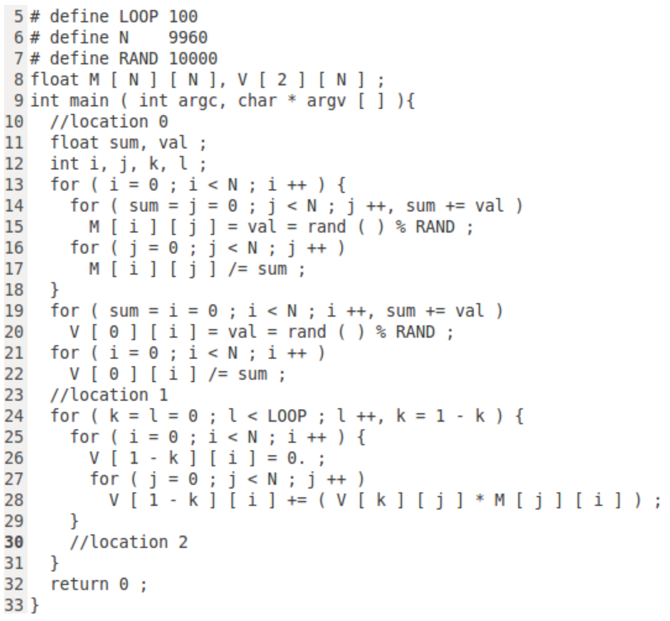
TABLE I
PRAGMAS OF DISCONTINUOUS INCREMENTAL CHECKPOINTING TECHNIQUE.

Pragma	Process	Monitor	
		Checkpointing mode	Recovering mode
start	send "start" to the monitor.	set all pages to read-only status.	find the next checkpoint: - if found: + inject checkpoint to the monitored process; + set all pages to read-only status; + if it is the last checkpoint: change to checkpointing mode; - else: notice error; stop process.
stop	send "stop" to the monitor.	back all pages to their original status.	back all pages to their original status.

save $< filename >$	send "save" to the monitor.	save the memory locations that have been modified to the current checkpoint;
		if $< filename >$ contains previous checkpoint:
		+ merge current checkpoint to $< filename >$;
		else:
		+ save current checkpoint to $< filename >$;
		set all pages to read-only status.
			notice error; stop process.

```
S1
# pragma dickpt start
S2₁
# pragma dickpt save <filename1>
S2₂
# pragma dickpt save <filename1>
# pragma dickpt stop
S3
# pragma dickpt start
S4
# pragma dickpt save <filename2>
# pragma dickpt stop
```

Fig. 2. Pseudo-code for discontinuous incremental checkpoints.



```
 5 # define LOOP 100
 6 # define N    9960
 7 # define RAND 10000
 8 float M [ N ] [ N ], V [ 2 ] [ N ] ;
 9 int main ( int argc, char * argv [ ] ){
10   //location 0
11   float sum, val ;
12   int i, j, k, l ;
13   for ( i = 0 ; i < N ; i ++ ) {
14     for ( sum = j = 0 ; j < N ; j ++, sum += val )
15       M [ i ] [ j ] = val = rand ( ) % RAND ;
16     for ( j = 0 ; j < N ; j ++ )
17       M [ i ] [ j ] /= sum ;
18   }
19   for ( sum = i = 0 ; i < N ; i ++, sum += val )
20     V [ 0 ] [ i ] = val = rand ( ) % RAND ;
21   for ( i = 0 ; i < N ; i ++ )
22     V [ 0 ] [ i ] /= sum ;
23   //location 1
24   for ( k = l = 0 ; l < LOOP ; l ++, k = 1 - k ) {
25     for ( i = 0 ; i < N ; i ++ ) {
26       V [ 1 - k ] [ i ] = 0. ;
27       for ( j = 0 ; j < N ; j ++ )
28         V [ 1 - k ] [ i ] += ( V [ k ] [ j ] * M [ j ] [ i ] ) ;
29     }
30     //location 2
31   }
32   return 0 ;
33 }
```

Fig. 3. Program computing the successive elements of a Markov Chain.

the second begins checkpointing at location 1 and does not take checkpoint at this location. For both cases, one hundred state vectors are computed at location 2 (line 30) and one checkpoint is generated after each computation. Our testbed is composed of an Intel Core2 Duo E8400 running at 3 GHz with 3 GB RAM and operated by Ubuntu 9.10 based on Linux kernel 2.6.31-21-generic. Table II presents the performance evaluation for four vector sizes (N equals to 3320, 6640, 9960 and 13280 elements respectively). For each vector size, performance are measured 30 times (mean values are provided in the table). In order to avoid the pollution of disk effects on measurements, all data (the virtual address space of processes, checkpoints, etc.) are resident in RAM.

The first section of Table II presents the size of checkpoints at location 1 (i.e. just after the initialization – as the case of traditional incremental checkpointer) and at location 2 (i.e. just after the computation of a new vector). This difference is the size of the transition matrix which is initialized at the beginning of the program. These data show how much disk space can be saved while abandoning the checkpoint at location 1 and, in the same way, how faster the checkpoint can be transfered over the network if necessary.

The second section of the table shows the time required to run the program without saving checkpoints, while saving all checkpoints and while saving location 2 checkpoints only. It highlights that the overhead involved by the generation of location 2 checkpoints is very light (between 1.5% and 3.3% for 100 checkpoints, i.e. between 0.01% and 0.03% per checkpoint) compared to the overhead involved by the generation of both location 1 and location 2 checkpoints (the total execution time is multiplied by up to 8.5).

The third section of the table provides the execution time to run the process restarting from loop iteration number 50.

TABLE II
PERFORMANCE EVALUATION.

	Matrix size			
	3320	6640	9960	13280
Checkpoint size (in MB)				
... at location 1	42.192	168.741	379.648	674.912
... at location 2	0.013	0.026	0.038	0.051
Total execution time (in seconds)				
... without generating checkpoints	11.34	41.30	108.27	168.69
... generating all checkpoints	13.10	58.14	393.60	1433.72
... generating location 2 checkpoints only	11.71	42.16	109.96	171.70
Execution time when restarting after loop iteration #50 (in seconds)				
... using location 1 and location 2 checkpoints	6.41	23.14	59.56	93.91
... using location 2 checkpoints only	6.09	21.84	56.64	88.71

Two cases are envisaged: the first one uses all checkpoints, i.e. the program is restarted, suspended at the beginning of function main, all checkpoints are injected in the process and the execution resumes at loop iteration 50; the second one uses location 2 checkpoints only, i.e. the program is restarted, suspended after the initialization step, all location 2 checkpoints are injected in the process and the execution resumes at loop iteration 50.

Performance measurements show that avoiding location 1 checkpoints is always beneficial.

C. Advantages and drawbacks

While comparing with the traditional incremental checkpointing technique, our new approach has the following strengths and weaknesses:

- In terms of performance. It reduces the execution time of the program for both checkpointing and recovering. It also strongly decreases the size of checkpoints.
- Regarding flexibility. It allows to select the segments to be checkpointed in programs. The case of normal incremental checkpointing is obtained by setting a `pragma dickpt start` and a `pragma dickpt stop` as the first and the last instruction respectively in the checkpointed program.
- More specifically for CAPE, it can directly extract and inject the `delta`, thus increasing the performance of this paradigm.
- Changing the source code when using a checkpointer is the most important drawback. Users have to insert directives to indicate the regions to be checkpointed. However, when used in CAPE, this insertion is done by the CAPE's compiler. As a result, this drawback has no impact for CAPE's users.
- Fragmentation of checkpoints: checkpoints which are not taken in a single block (surrounded by a pair of `# pragma dickpt start` and `# pragma dickpt stop)` can not be merged to unique checkpoint. So, many files are needed to contain the checkpoints of the different checkpointing blocks. This drawback is important when checkpoints reference the same memory area.

IV. CHECKPOINT STRUCTURE OPTIMIZATION

The structure of a complete checkpoint is usually quite straightforward. After some very specific data like the content of registers and the size of the memory, the rest of a complete checkpoint is usually composed of the content of all the memory pages stored the one after the other one.

In the case of an incremental checkpoint, several cases have to be envisaged. Alike complete checkpoints, the content of registers are also stored with incremental checkpoints. However, the case of memory updates is different. The solution really depends upon the granularity of data, which ranges from one byte to one page with the most interesting case probably at one word.

Memory granularity

There are two main drawbacks when the granularity is the page. The first one is that a complete page must be saved even though a single byte in the page has been modified, which is not really memory efficient. Considering the size of today disks, this may not be a problem unless a very large number of checkpoints have to be generated. The problem may have a more important impact if for examples these checkpoints have to be sent over the network, especially with a limited bandwidth. The second main drawback is that it provides no information on which bytes in the page have been modified effectively. The latter drawback definitively forbids any merge operation of successive incremental checkpoints.

Setting the granularity of the checkpoint to a single byte solves the memory inefficiency problem of the page granularity. However, it leads to other subtle problems, like for example the reference to memory locations that do not exist in the virtual address space of the process. Let $< a, b, c, d >$ be four consecutive bytes stored at a memory location and representing a pointer to another memory location. After a first checkpoint, this memory location may contain $< a, b', c, d >$. After a second checkpoint, the same memory location may contain $< a, b, c', d >$. If, for any reasons, it is required to merge the two checkpoints (and this is typically the case with CAPE), the result may lead to $< a, b', c', d >$ which may not be part of the virtual address space of the process.

Setting the granularity of the checkpoint to a word (i.e. four bytes) is the best compromise as it solves the problem of memory space efficiency and does not introduce any pointer problem as described above. This solution is not the perfect solution, especially for string of characters. However, problems involved by setting the granularity of the checkpoint to a word has no significant impact on the execution of the program.

Finally, one can note that setting the granularity of the checkpoint to the entire virtual address space turns an incremental checkpointer into a complete checkpointer.

Incremental checkpoint content

Apart from the specific values also stored in a complete checkpoint, an incremental checkpoint should be composed of the list of memory locations that have been modified since the beginning of the execution of the program, or since the last checkpoint, and the current value for each of these specific memory locations. The simplest way to store a such list is to save both the address and its associated value for all modified memory locations the one after the other one. However, since the spatial locality of data in most programs implies that a modification at a memory location increases the probability for adjacent memory locations to be modified as well, this way of storing data is not necessarily efficient.

Thus, in order to take advantage of the spatial locality of updates and therefore reduce the size of checkpoints, several alternative methods for storing memory updates have been identified:

- *Single data*. This case occurs when a single memory location has been updated. In this case, the only information to store are the address of the memory location and the content at the memory location. Data to store into the checkpoint are:
$$< \text{addr}, \text{value} >$$
- *Several successive data*. This case occurs when more than one consecutive memory locations have been updated. For example, this is encountered when the content of an array has been modified. The best way to store all the information in this case is:
$$< \text{addr}, \text{size}[, \text{value...}] >$$
- *Many data*. This occurs when lots of non-successive memory locations have been updated on a single page. In this case, instead of storing a large number of Single data and Several successive data elements, it is more efficient to store the address of the page, the list of memory locations on the page that have been modified and for each modified memory location the associated value. The efficiency of this solution resides in the mapping, i.e. the list of memory locations on the page. As this is a binary information for each data in the page, it can be represented using a single bit per memory location. For example, for a 4-kB page, the size of the map is 1024 bits (or 128 bytes) with a granularity set a word.
$$< \text{addr}, \text{map}[, \text{value...}] >$$
- *Entire page*. This occurs when all memory locations on a memory page have been modified. This case is quite common when a new page is added to the virtual address space of a process. The best way to store the complete content of a page is:
$$< \text{addr}[, \text{value...}] >$$
No size need to be provided in this case as it is implicitly known.

Table III compares the amount of memory needed to store updated data for all cases presented above. The size of a memory page is assumed to be 4 kB. Let a *chunk* be a set of contiguous memory locations that have been updated. Let c be the number of chunks in a memory page, let s_i be the number of elements in chunk i and let u be the number of updates in the memory page. By definition, $\sum_{i=1}^{c} s_i = u$.

TABLE III
AMOUNT OF MEMORY TO STORE UPDATES.

Method	Amount of memory for a single chunk	for a page
Single data (SD)	8	$8 \times u$
Several successive data (SSD)	$8 + 4 \times s$	$8 \times c + 4 \times u$
Many data (MD)	$132 + 4 \times u$	$132 + 4 \times u$
Entire page (EP)	4100	4100

Fig. 4 shows a comparison of the amount of memory needed to store all updates in a 4-kB page as a function of the number of updated memory locations in the page. SD, MD and EP only depend upon the number of updated memory locations while SSD also depends on the distribution of the updated memory locations. As a result, Fig. 4 shows both the best case (SSD_{min}) that is when all updated memory locations are in a single chunk, and the worst case (SSD_{max}) that is the case when updated memory locations are distributed in the configuration that requires the maximum number of chunks. For a 4-kB memory page and a 4-byte word, this maximum is given by:

$$\begin{cases} \lfloor u/2 \rfloor & \text{if } 0 < u \leq 682 \\ 1024 - u & \text{if } 682 < u \leq 1024 \end{cases}$$

One can note that when two successive memory locations have to be stored, the amount of memory needed to store the information for both Single data and Several successive data cases is the same.

The most efficient solution, i.e. the one that reduces the memory usage the most, is identified in this way. For each page, first the Many data representation is built. It requires at most 4228 bytes; second, a combination of Single data and Several successive data methods is built, having Single data chosen for isolated data and Several successive data chosen when at least two consecutive memory locations have been updated; third, the shortest representation between both computed is stored. Note that the Entire page method is left for the storage of a new page.

From the expressions provided in Table III one can demonstrate that the Several successive data method is always the most interesting solution when the number of updates is smaller than 34. Then, there is a trade-off between the

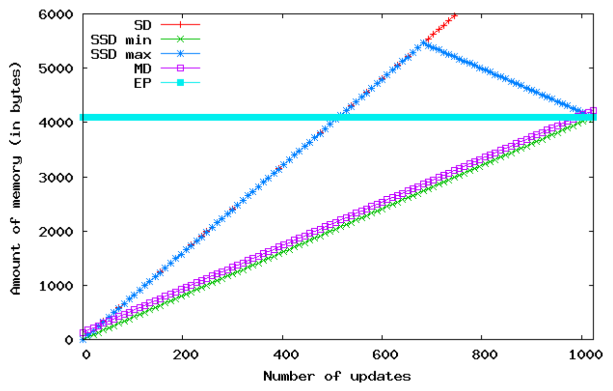


Fig. 4. Amount of memory to store updates.

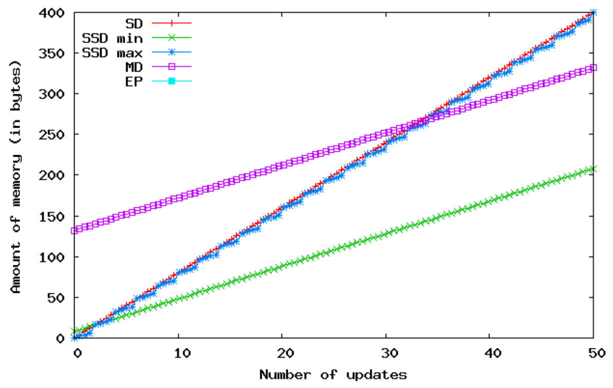


Fig. 5. Trade-off between SSD and MD.

Several successive data method and the Many data method that depends on the number of chunks. Fig. 5 is a magnification of Fig. 4 for a number of updates in the range from 0 to 50.

Identifying the method

Considering that more than one method is used to store memory updates, it is important to identify which one was used when restoring the content of the checkpoint. A simple solution would have consisted in adding an extra integer or even a byte before any data description or set of data description. However, in order to keep the size of the checkpoint as small as possible, it has been decided to add no extra byte to the checkpoint.

Instead, considering that all methods require an address as the first field and that these addresses are necessarily aligned on a boundary of a word, i.e. these addresses are necessarily a multiple of 4 or the last two digits of their binary representation are necessarily 00, it is possible to use this ”free” space to store which method was used to store the data. In our current implementation, 00 is associated to Single data, 01 to Several successive data, 10 to Many data and 11 to Entire page. When restoring the content of a checkpoint, these two bits are reset to 00 after the storage method has been identified and before the address is effectively used.

V. Conclusion and Future Works

For the development of our distributed implementation of OpenMP using the CAPE paradigm, an efficient incremental checkpointer is required. This article presented our new approach called discontinuous incremental checkpointing. The initial implementation of this approach has proved its efficiency in terms of size – the amount of memory space required to store the checkpoints is significantly reduced – and time to generate the checkpoints, to restart from a checkpoint or to send the checkpoint over the network. This approach led to very lightweight checkpoints, the size being in the order of a few kB while the size of the virtual address space of a process is in the order of tens or hundreds of MB. However, after the significant modification in the possibility and the nature of the checkpointer, new paradigms may be developed for CAPE and we are investigating in this way.

Apart from processor registers, the current implementation of our incremental checkpointer does not save system information about the process, like open file descriptors, sockets, POSIX semaphore or shared memory, etc. Somehow, this is not a real drawback as if such declarations are set outside of checkpointed segments, they could be re-executed in case of restoration. However, in the near future, these system data should be included and the overhead involved by their inclusion of each of them will be studied. In fine, we would like this incremental checkpointer to be completely customizable so that users would be able to store only relevant information.

References

[1] http://www.research.rutgers.edu/ edpin/epckpt/

[2] J. Ansel, K. Arya and G. Cooperman. *DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop*. 32rd IEEE International Parallel and Distributed Processing Symposium (IPDPS09), Rome, Italy, May 2009.

[3] http://cryopid.berlios.de/

[4] http://pages.cs.wisc.edu/ zandy/ckpt/

[5] http://hpc.pnl.gov/sft/tick.html

[6] S. YI, J. Heo, Y. Cho, J. Hong, J. Choi and G. Jeon. *Ickpt: An Efficient Incremental Checkpointing Using Page Writing Fault - Focusing on the Implementation in Linux Kernel*. Proceedings of the ISCA 19th International Conference on Computers and Their Applications (CATA04), Seattle, WA, pp. 209-212, March 2004.

[7] http://openmp.org/wp/

[8] E. Renault. *Parallelization of For Loops Using Checkpointing Techniques*. Proceedings of the 2005 International Conference on Parallel Processing Workshops, Oslo, Norway, pp 313-319, June 2005.

[9] E. Renault. *Distributed Implementation of OpenMP Based on Checkpointing Aided Parallel Execution*. International Workshop on OpenMP (IWOMP), Beijing, China, LNCS 4935, pp 183-193, June 2007.

[10] L. Mereuta and E. Renault. *Checkpointing Aided Parallel Execution Model and Analysis*. High Performance Computation Conference (HPCC), Houston, TX, LNCS 4782, pp 707-717, September 2007.

[11] J.S. Plank, M. Beck, G. Kingsley and K. Li. Libckpt: Transparent Checkpointing under Unix. Proceedings of the Usenix Winter 1995 Technical Conference, New Orleans, LA, pp. 213-223, January 1995.

[12] http://checkpointing.psnc.pl/Progress/psncCR/

[13] O.O. Sudako, I.S. Meshcheriakov, Y.V. Taras ShevchentoA. *Transparent Checkpointing System for Linux Clusters*. 4th IEEE worksho on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS). 2007.

[14] Y.-M. Wang, Y. Huang, K.-P. Vo, P.-Y. Chung, and C. Kintala. *Checkpointing and its applications*. 25th Annual Intl symposium on Fault-Tolerant Computing, pp.22-30, Oct. 1995.