This article appeared in Journal of Systems Architecture
vol.53, no.9, pp.659-676, 2007

Multiprogrammed Non-blocking Checkpoints in Support of Optimistic Simulation on Myrinet Clusters*†

Andrea Santoro and Francesco Quaglia
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy

Abstract

CCL (Checkpointing and Communication Library) is a software layer in support of optimistic Parallel Discrete Event Simulation (PDES) on myrinet-based COTS clusters. Beyond classical low latency message delivery functionalities, this library implements CPU offloaded, non-blocking (asynchronous) checkpointing functionalities based on data transfer capabilities provided by a programmable DMA engine on board of myrinet network cards. These functionalities are unique since optimistic simulation systems conventionally rely on checkpointing implemented as a synchronous, CPU-based data copy. Releases of CCL up to v2.4 only support monoprogrammed non-blocking checkpoints. This forces re-synchronization between CPU and DMA activities, which is a potential source of overhead, each time a new checkpoint request must be issued at the simulation application level while the last issued one is still being carried out by the DMA engine. In this paper we present a redesigned release of CCL (v3.0) that, exploiting hardware capabilities of more advanced myrinet clusters, supports multiprogrammed non-blocking checkpoints. The multiprogrammed approach allows higher degree of concurrency between checkpointing and other simulation specific operations carried out by the CPU, with benefits on performance. We also report the results of the experimental evaluation of those benefits for the case of a Personal Communication System (PCS) simulation application, selected as a real world test-bed.

Keywords: Optimistic Simulation, Rollback-based Synchronization, Checkpointing, DMA, Myrinet, Performance Optimization, Multiprogramming.

*An earlier version of this article, with title "CCL v3.0: Multiprogrammed Semi-Asynchronous Checkpoints", appeared in *Proc. of the 17th ACM/IEEE/SCS Workshop on Parallel and Distributed simulation (PADS'03)*.

†Information on the CCL project and free software releases are available at the URL http://www.dis.uniroma1.it/PADS/software/CCL/

1 Introduction

CCL (Checkpointing and Communication Library) is a software project [19, 20] in support of optimistic Parallel Discrete Event Simulation (PDES) [8, 10] on myrinet-based Commercial-Off-The-Shelf (COTS) clusters. These are commonly recognized target platforms for parallel computing applications. CCL offers both classical low latency message delivery functionalities, implemented according to standard mechanisms tailored for the myrinet architecture [16], and innovative non-blocking, DMA-based checkpointing functionalities that allow the task of checkpointing the Logical Process (LP) state vector to be offloaded from the CPU.

A core mechanism CCL relies on is re-synchronization between simulation activities carried out by the CPU and non-blocking checkpointing activities carried out through DMA. Such a mechanism has been employed to prevent both (i) data inconsistency, that might arise whenever the LP state vector currently being checkpointed through DMA is accessed by the LP for further modifications of some state variables, and (ii) contention on the myrinet device due to checkpoint requests that would be issued at the simulation application level while the last accepted one is still being handled by the device itself.

In releases of CCL up to v2.4, re-synchronization implementations have relied on threshold-based mechanisms [20, 22]. According to these mechanisms, an on-going non-blocking checkpoint operation is either committed or aborted upon re-synchronization occurrence depending on the current advancement state of the operation itself. In case of abort, there is an interruption of any in-progress DMA-based data transfer associated with the operation. In case of commit, the remaining part (if any) of the data transfer between the state vector buffer and the checkpoint buffer is charged to the CPU, so to exploit the higher transfer rate on the system bus as compared to the one proper of the PCI bus used by the Myrinet DMA device. Proper selection of the threshold values has been shown to allow adequate balance between the CPU cost of committing an on-going checkpoint operation (i.e. completing the data transfer) upon re-synchronization and the additional expected state recovery cost due to uncommitment of checkpoints ([1]).

In this paper we present a redesigned and reengineered version of CCL (v3.0) which allows any LP to issue a checkpoint request independently of the fact that the myrinet device is still handling an already accepted request. In other words this version supports *multiprogrammed* non-blocking checkpoints as opposed to releases up to v2.4, which follow a monoprogrammed

[1] In case recovery to an uncheckpointed state vector value is required, e.g. because that checkpoint was aborted, such a value needs to be reconstructed by reloading into the LP state vector buffer the latest checkpoint preceding the state to be recovered and reprocessing intermediate events. The reprocessing phase is typically termed *coasting forward* and gives rise to overhead in the recovery procedure.

approach.

The reason why the multiprogrammed approach was not embedded in those releases primarily resides in the reduced computational power of the hardware they have been designed for, namely M2M-PCI32C myrinet cards. Actually this type of card is equipped with a 33 MHz RISC processor, therefore management of any support operation for multiprogramming (e.g. checkpoint requests queuing/dequeuing) at this relatively low processor speed might significantly reduce the responsiveness of CCL in carrying out performance critical tasks, such as communication. CCL v3.0 has been developed for the more recent M3M-PCI64C myrinet card, based on a 200 MHz RISC processor and on a set of hardware features that even facilitate the management of multiprogrammed checkpoints.

With respect to the relevance of multiprogrammed non-blocking checkpoints, they actually allow a reduction of the impact of re-synchronization since the re-synchronization functionality must be invoked exclusively to maintain data consistency, i.e. when an LP is scheduled for event execution while its state vector is still being checkpointed through DMA (this cannot be avoided unless we accept the risk of incorrect state recovery when using that checkpoint during rollback). As a consequence, the multiprogrammed approach permits higher degree of concurrency between checkpointing and other simulation specific operations carried out by the CPU, with obvious positive effects on the performance of the simulation system. These effects have been evaluated for the case of a Personal Communication System (PCS) simulation application, selected as a real world test-bed, and the related experimental results are also reported in this paper.

The remainder of this work is structured as follows. In Section 2 we provide a short, technical description of the CCL v2.4, reporting details on the M2M-PCI32C myrinet card. This will form the basis for the description and the understanding of the improved multiprogrammed approach. In Section 3 we present the main hardware differences between M2M-PCI32C and M3M-PCI64C cards, and the software supports for multiprogrammed non-blocking checkpoints. Actually, while presenting those supports we discuss/justify specific design choices, and we also report, whenever required, experimental data demonstrating their effectiveness. Related work is overviewed in Section 4. The results of the experimental study on the PCS simulation application are reported in Section 5.

2 CCL v2.4

2.1 Underlying Hardware

As already mentioned, CCL v2.4 has been designed for M2M-PCI32C myrinet network cards relying on the LANai 4 chip [13]. These cards consist of the following main components (see Figure 1):

(A) An internal bus, namely LBUS (Local BUS), clocked at 66 MHz, i.e. twice the chip clock speed.

(B) A programmable RISC processor connected to the LBUS, which we will refer to as LANai processor.

(C) A RAM bank of 1 Mbyte (LANai internal memory), connected to the LBUS, which can be mapped into the memory address space of the host.

(D) A packet interface between the myrinet switch and the LANai chip, accessible by the LANai processor.

(E) Three DMA engines used for:

  (i) packet-interface/internal-memory transfer (Receive DMA),

  (ii) internal-memory/packet-interface transfer (Send DMA),

  (iii) internal-memory/host-memory transfer or vice-versa (EBUS DMA, namely External Bus DMA).

Host access to the LANai internal memory takes place through a PCI bridge, which is also used for EBUS DMA data transfer from the host memory to the LANai internal memory and vice versa. The LANai processor has the responsibility to activate the three DMA engines (these engines cannot be activated by the host CPU). A DMA operation can be activated by the LANai processor only after the last one (on the same DMA) is already completed. In other words there is no possibility to pre-define a batch of operations to be executed by any DMA engine with no intervention of the LANai processor.

2.2 Software Features

Communication functionalities are implemented in CCL according to common solutions supporting fast speed messaging on the myrinet architecture [16]. Messages incoming from the network are temporarily buffered in the LANai internal memory (data transfer between the

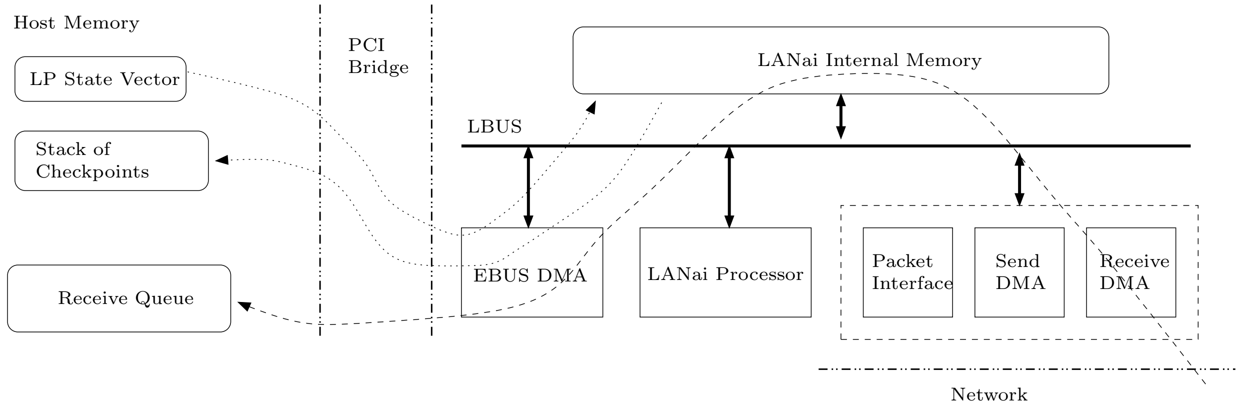

Figure 1: Basic Architecture of a Myrinet Card.

packet interface and the internal memory takes place through the Receive DMA) and then transferred into the receive queue, located onto host memory, through the EBUS DMA (see the directed dashed line in Figure 1).

Following another common design choice, any send operation issued by the application involves copying the message content directly into the LANai internal memory. This is also referred to as "zero-copy" send. Then the message is transferred onto the network through the Send DMA. This optimization allows keeping the delivery latency at a minimum by avoiding intermediate buffering (e.g. in kernel memory) at the sender side.

The EBUS DMA is used not only to transfer messages from the LANai internal memory to the receive queue, but also to perform data transfer associated with checkpointing. Specifically, a checkpoint operation involves data transfer from the LP state buffer (located onto host memory) to the stack of the checkpointed states of the LP (also located onto host memory). As shown by the directed dotted lines in Figure 1, the transfer operation is charged to the EBUS DMA that uses the LANai internal memory as an intermediate buffer ([2]).

The LANai processor has the responsibility to program the three DMA engines anytime there is the need for supporting a given data transfer operation. This is done through a proper *control program* run by the LANai processor. Therefore, issuing a checkpoint request at the application level actually means requesting the LANai processor to program the EBUS DMA for the data transfer operation. This takes place by writing into the LANai internal memory all the information required by the LANai processor to program the EBUS DMA, i.e. the addresses of the LP state buffer and of the checkpoint stack entry. Such a writing is per-

[2]Intermediate buffering is needed since the EBUS DMA does not support host-memory to host-memory data transfer directly. It only supports host-memory to internal-memory transfer or vice versa.

formed through the API function `non_block_ckpt(int LP_id, double simulation_clock)`, where `LP_id` is the identifier of the LP whose state vector needs to be checkpointed, and `simulation_clock` is the value of the current simulation time of the LP ([3]).

Any checkpoint operation is split by the LANai control program into a sequence of data transfer operations to be performed by the EBUS DMA. Each operation transfers up to a maximum amount of bytes, called *burst*, from the LP state vector to the LANai internal memory (intermediate buffering) or from the LANai internal memory to the stack of checkpoints of the LP. Also, lower priority is assigned by the control program to data transfer associated with checkpointing as compared to message transfer into the receive queue. This engineering choice allows checkpointing functionalities offered by CCL to produce negligible interference with communication functionalities, namely the primary task to be carried out by the network card. Specifically, splitting any checkpoint operation into a sequence of bursts allows prompt re-assignment of the hardware resources on board of the myrinet card (i.e. the EBUS DMA, the PCI bridge and the LBUS) to communication operations due to their higher priority. As respect to this point, in [21] it has been shown how to identify the maximum value for the burst length which allows the performance of communication functionalities not to be significantly perturbed by the activation of checkpointing functionalities. The use of such a maximum value is recommended since it keeps the checkpoint latency at a minimum by keeping low the amount of bursts required for the operation itself.

As already noted, re-synchronization between CPU activities and checkpointing activities carried out by the EBUS DMA is used to avoid data inconsistency and to prevent contention on the myrinet hardware. The re-synchronization functionality offered by CCL v2.4 relies on a threshold-based mechanism [19]. This is implemented by maintaining a counter, namely `completed_transfers`, in the LANai internal memory, which is managed as follows. The counter is reset by the function `non_block_ckpt()` upon issuing a checkpoint request at the application level. It is incremented by the LANai control program each time the program itself becomes aware that an EBUS DMA data transfer (from/to host memory) associated with the checkpoint operation has been completed. The value of this counter is used by the re-synchronization function `ckpt_cond_abort(float threshold)`, where the parameter `threshold` indicates the advancement percentage of the last activated checkpoint operation, if any, under which the operation itself must be aborted upon re-synchronizing. To compute such an advancement percentage, `ckpt_cond_abort()` uses information maintained by an additional counter, namely `total_transfers`, which records the total number of EBUS DMA

[3]CCL manages the checkpoint stacks of the LPs in a totally transparent way to the application programmer, which is the reason why `LP_id` is a sufficient parameter to identify both the state buffer and the entry into the stack of checkpoints that must be involved in the data transfer.

transfers (from/to host memory) to complete the checkpoint operation. This counter value is set upon issuing the checkpoint request as a function of the LP state vector size and the length of the burst. If $\frac{\texttt{completed_transfers}}{\texttt{total_transfers}} < \texttt{threshold}$, then the checkpoint operation is aborted. This is done by setting a flag, namely `ckpt_abort`, located in the LANai internal memory which indicates to the LANai control program that the EBUS DMA does not need to be programmed for the current checkpoint operation anymore (i.e. no additional burst related to that operation must be carried out). Otherwise, software at the host side (i.e. the re-synchronization function `ckpt_cond_abort()`), beyond notifying to the control program that no additional burst needs to be carried out, also completes the checkpoint operation by performing a `memcpy()` that transfers the remaining portion of the LP state vector being checkpointed into the checkpoint buffer. Actually, the fact that the remaining portion of the checkpoint operation upon re-synchronization is carried out through the CPU results in an optimization of the completion latency of the operation itself since the CPU works on the system bus, which typically exhibits higher transfer rate with respect to the PCI bus used by the EBUS DMA (see [25] for a quantitative evaluation).

3 CCL v3.0

3.1 Underlying Hardware

M3M-PCI64C myrinet cards [15], based on the LANai 9 chip [14], are equipped with components that are similar to those on board of LANai 4, namely the LBUS, the three DMA engines, the internal memory, the packet interface and the RISC processor. Beyond LBUS/processor speed (400/200 MHz on LANai 9 vs 66/33 MHz on LANai 4), a main difference between cards based on LANai 4 and LANai 9 resides in the EBUS DMA controller. Specifically, in the LANai 4 chip information related to a single EBUS DMA operation must be explicitly passed by the LANai processor to the DMA controller upon the activation of the DMA operation. Instead the controller on board of cards employing LANai 9 uses chains of *control blocks* stored in the LANai internal memory to activate DMA operations. The structure of a control block for DMA operations is as follows:

```
struct DMA_BLOCK{
    volatile uint32 next; /* next block in chain pointer */
    uint16 spare;         /* unused */
    uint16 csum;          /* ones complement cksum of this block */
    uint32 len;           /* byte count */
    uint32 lar;           /* lanai address */
    uint32 eah;           /* high PCI address */
```

```
    uint32 eal;            /* low PCI address */
}
```

The low-order bits of the `next` pointer are used as flags to determine the DMA direction and whether this is the last block in the chain, according to the following specification:

bit	function
0	Direction (1=host→LANai, 0=LANai→host)
1	Terminal (1=terminal, 0=not-terminal)

The terminal bit indicates that this is the end of the chain. If the terminal bit is already set when examined by the DMA controller, the operation specified by that block is not executed, and the controller stops moving along the chain.

EBUS DMA operations can be split into 4 channels (i.e. 4 chains of control blocks), with different priority levels. The DMA controller holds 4 pointer registers that maintain the chain base addresses. To use a particular channel of the DMA controller, the chain base address register must be initialized to a LANai memory location that will be used to hold a DMA control block. Once the chain base address has been written, and the control block has been properly initialized, the DMA can be triggered from the host or from the LANai processor. Actually the LANai processor has control (i.e. it can trigger DMA) on 2 channels only, namely channel 0 and channel 2, with channel 0 having higher priority on channel 2.

A chain is terminated by either a control block with its terminal bit set (this is the case discussed above), or a control block with the `next` pointer set to 0. In both cases the control block is referred to as *terminal block*. The DMA engine will execute each block in the chain until it sees a terminal block. It will not perform the DMA operation described by that block. Instead, it will go to sleep on that channel until the `next` pointer of that terminal block is altered to make it valid, and the channel is reactivated. Multiple re-activations when a channel is already active are harmless. When the DMA for a block is completed, the DMA engine will automatically set the terminal bit of that block. The only exception to the processing of a non-terminal block is when the `len` field is set to 0, in which case nothing is done and the DMA controller advances to the next block.

3.2 Management of the EBUS DMA

As seen in Section 2.2, in a classical message passing organization for myrinet the EBUS DMA is responsible for transferring messages incoming from the network into the receive queue. We retain this approach in CCL v3.0 according to the following implementation.

A vector of DMA control blocks, namely `message_DMA[]`, is maintained in the LANai internal memory. Each entry keeps information related to a DMA operation for transferring

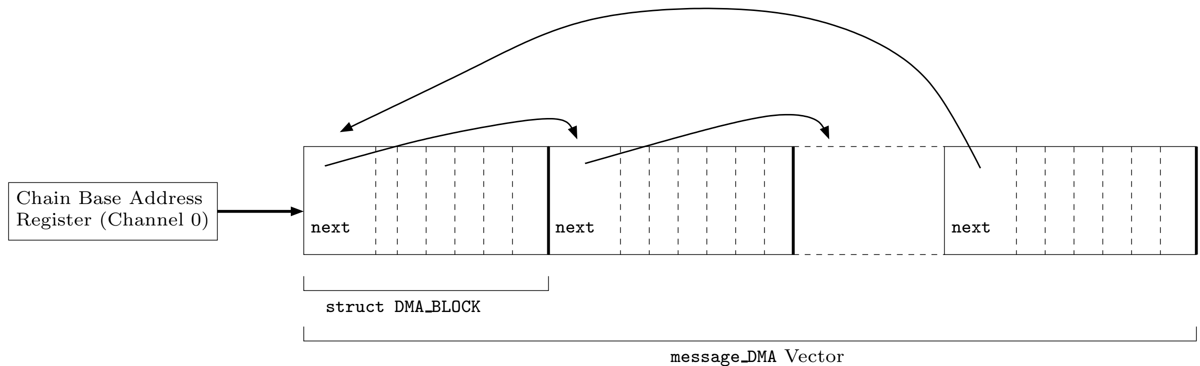

Figure 2: message_DMA[] Vector.

messages incoming from the network into the receive queue. (Recall these messages are transferred from the network in the LANai internal memory through the receive DMA). The chain base address register associated with the DMA channel 0 points to the first entry of message_DMA[] (see Figure 2). Each entry, but the last one, has the next field pointing to the successive entry in the vector. Instead, the next field of the last entry of message_DMA[] points to the first entry of the same vector. In other words, the message_DMA[] vector is managed as a circular buffer of DMA control blocks.

Intially, no control block in message_DMA[] is valid, therefore the terminal bit associated with each entry is set. This also means that all the entries of the message_DMA[] vector are available to the LANai control program for packing DMA transfer operations of incoming messages into the receive queue. However, given that the data transfer direction for those DMA operations is fixed (LANai to host), the direction bit is adequately set in all the entries of the vector.

When a DMA operation is required for transferring a message into the receive queue, the LANai control program adequately sets the fields (e.g. len and lar) of the first available entry in message_DMA[], resets the terminal bit in that entry and then triggers the EBUS DMA controller on channel 0. (We recall that in case the controller is already active on that channel, triggering is harmless anyway.) Then the index of the first available entry of message_DMA[] is increased modulo the size of the vector. When the DMA operation is completed, the EBUS DMA controller automatically sets the terminal bit associated with the corresponding entry of the message_DMA[] vector, thus making that entry available again.

As for CCL v2.4, in CCL v3.0 the EBUS DMA also has the responsibility to perform data transfer associated with checkpointing. In order not to harm communication performance, message transfer into the receive queue must have higher priority as compared to bursts of data transfer associated with checkpointing [21]. To maintain this basic design feature while

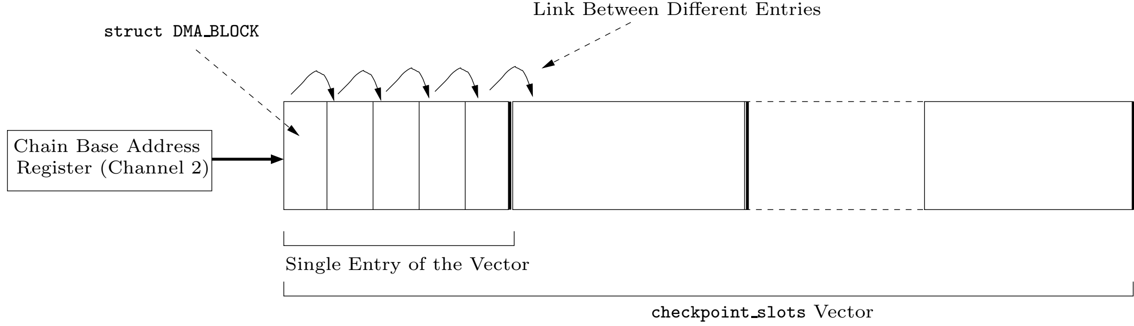

Figure 3: `checkpoint_slots[]` Vector.

allowing multiprogrammed non-blocking checkpoints, we exploit the second DMA channel, namely channel 2, that can be triggered by the control program run by the LANai processor. Recall this channel has lower priority as compared to channel 0 used for transferring messages from the LANai internal memory into the receive queue.

We maintain in the LANai internal memory a second vector, namely `checkpoint_slots[]`, whose structure is shown in Figure 3. Each entry keeps the information required by the EBUS DMA for any single checkpoint operation. According to the classical CCL design choice, each checkpoint operation is implemented as a sequence of bursts that transfer pieces of the LP state vector in the LANai internal memory (intermediate buffering) and then into the checkpoint stack located on host memory. Therefore, each entry of the `checkpoint_slots[]` vector maintains information related to all the bursts required for the checkpoint operation, e.g. memory addresses and data transfer direction. To achieve this, an entry of `checkpoint_slots[]` is structured as a sequence (i.e. a vector) of DMA control blocks, each one maintaining information related to a single burst. The size of the entries, namely the amount of DMA control blocks within the entry, can be determined at compile time of CCL as a function of the maximum LP state vector size and the length of the burst. This size must be equal to the maximum amount of bursts required for a checkpoint operation, which can be derived by the maximum LP state vector size and the burst length according to the following expression:

$$MAX_BURSTS = 2 \times \lceil \frac{maximum\ state\ vector\ size}{burst\ length} \rceil \quad (1)$$

where the multiplier factor two accounts for intermediate buffering of pieces of the LP state vector in the LANai internal memory.

DMA control blocks within each entry of the `checkpoint_slots[]` vector are linked as a list. Also, the last block of the entry has the `next` field pointing to the first block of the successive entry in the vector. In other words, at a memory layout level `checkpoint_slots[]`

is structured as a linked list of DMA control blocks, each one maintaining information related to a single burst of data transfer associated with checkpointing. The blocks within the same entry of `checkpoint_slots[]` are associated with the same checkpoint operation.

As for the `message_DMA[]` vector, the last DMA control block of the last entry of the vector `checkpoint_slots[]` points to the first DMA control block of the first entry of the same vector so that circular management of the vector entries is performed. Finally, the chain base address register associated with the DMA channel 2 points to the first control block of the first entry of `checkpoint_slots[]`.

Similarly to what is done for the `message_DMA[]` vector, all the DMA control blocks in any entry of `checkpoint_slots[]` have the terminal bit initially set, which indicates that no control block currently maintains valid information for a burst associated with checkpointing. When a non-blocking checkpoint operation is issued at the application level through the function `non_block_ckpt()`, the memory addresses of the LP state buffer and of the checkpoint stack entry are notified to the control program run by the LANai processor which then adequately sets the fields of all the DMA control blocks in the first available entry of the `checkpoint_slots[]` vector, resets the terminal bits in all those blocks, and triggers the EBUS DMA controller on channel 2. Then the index of the first available entry of the `checkpoint_slots[]` vector is increased modulo the size of the vector. Given that the terminal bits of DMA control blocks are automatically set by the EBUS DMA controller as soon as the corresponding DMA operation is completed, an entry of `checkpoint_slots[]` is made available again as soon as all the DMA control blocks within that entry have been executed by the EBUS DMA, i.e. when the checkpoint operation has been completed.

Since the size MAX_BURSTS of the entries of `checkpoint_slots[]` is determined according to expression (1) as a function of the maximum LP state vector size, there is the possibility that a checkpoint operation of a state vector with size less that the maximum one actually needs a number of bursts X that is less than MAX_BURSTS. In this case, upon packing the checkpoint operation within a given entry of `checkpoint_slots[]`, the LANai control program adequately sets the `next` field of the X-th DMA control block in that entry in order to make it point to the first control block of the successive entry of the vector. In other words, the last $(MAX_BURSTS - X)$ DMA control blocks within the entry of `checkpoint_slots[]` that will contain the information related to the checkpoint operation are not visited while following the linked list.

We note that also the number of entries of the `checkpoint_slots[]` vector can be established at compile time. Hence, given that for data consistency issues, re-sychronization needs to occur before re-scheduling each LP, and given that (as we will also discuss in detail in

Section 3.4) a new checkpoint operation is issued for an LP only after it has been re-scheduled for event execution, reserving within `checkpoint_slots[]` a number of entries greater than or equal to the nunber of LPs on a machine prevents running out of entries.

Overall, we have implemented a mechanism that allows multiprogrammed non-blocking checkpoints through a batch of checkpoint operations (packed within the entries of the vector `checkpoint_slots[]`). Bursts associated with checkpoint operations already in the batch are automatically executed after each other, with no additional task to be performed by the LANai control program. This takes place thanks to capabilities of the EBUS DMA controller which automatically searches for EBUS DMA operations along the list of DMA control blocks associated with channel 2 of the EBUS DMA.

Discussion on v2.4 vs v3.0. When a checkpoint request is issued at the application level, a similar amount of work must be performed by the LANai processor in CCL v2.4 and v3.0 while accepting the request. Specifically, in both versions, the function `non_block_ckpt()` notifies to the LANai control program the memory addresses of the LP state buffer and of the checkpoint stack entry. Then the LANai control program computes the bursts required for the operation (e.g. the number of bursts and the corresponding memory addresses) and writes the outcome into a proper location in the LANai internal memory. For CCL v3.0, such a location corresponds to an entry in the `checkpoint_slots[]` vector.

On the other hand, by exploiting innovative hardware features, some work related to the management of the checkpoint operation has been even offloaded from the LANai processor in CCL v3.0. Specifically, the EBUS DMA controller on cards based on the LANai 9 chip automatically handles priorities between DMA operations associated with message transfer into the receive queue and bursts associated with checkpointing, thus offloading priority management from the control program. Also, the control program does not need to poll on the EBUS DMA to verify whether the last activated DMA operation has been completed prior to activate a new operation. Polling is used in CCL v2.4 since, as seen in Section 1, no batch of EBUS DMA operations can be pre-defined.

Finally, dequeuing already executed checkpoint operations from the batch does not impose additional work on the LANai control program since the terminal bits of the DMA control blocks within any entry of the `checkpoint_slots[]` vector are automatically set by the EBUS DMA controller. This automatically removes that entry from the batch since the EBUS DMA will not execute control blocks in that entry anymore unless a new checkpoint operation is packed in the same entry.

3.3 Re-synchronization

The re-synchronization function for CCL v3.0 implements the same threshold-based mechanism as v2.4, and has the prototype `ckpt_cond_abort(int LP_id, float threshold)`. As compared to CCL v2.4, the parameter `LP_id` has been introduced since, according to the multiprogramming mechanism described in the previous section, multiple checkpoint requests from distinct LPs can be included in the batch and serially processed. Therefore when calling the re-synchronization function we must define for which LP re-synchronization must really occur, i.e. for which checkpoint operation in the batch a commitment/abort decision must be taken. Other checkpoint operations within the batch, associated with different LPs, are unaffected, i.e. they remain within the batch for serial processing.

To support re-synchronization in such a selective way, we have used the following data structures:

`LP_ids[]`. This is a vector of integers maintained in the LANai internal memory, which has the same size of `checkpoint_slots[]`. `LP_ids[i]` keeps track of the LP identifier associated with the checkpoint operation packed in `checkpoint_slots[i]`. Its value is set by the LANai control program upon accepting a checkpoint request and packing the corresponding checkpoint operation into `checkpoint_slots[i]`.

`total_transfers[]`. This is a vector of integers maintained in host memory. The j-th entry of this vector keeps track of the total amount of bursts required for a checkpoint operation involving the state vector of the j-th LP. Its value is set by the function `non_block_ckpt()` when a non-blocking checkpoint operation is issued by the j-th LP.

`completed_transfers[]`. This is a vector of integers maintained in the LANai internal memory. The j-th entry of this vector keeps track of the total amount of bursts already executed for the last activated checkpoint operation involving the state vector of the j-th LP. It is inizialized to zero by the function `non_block_ckpt()` when a non-blocking checkpoint operation is issued by the j-th LP. It is incremented by one each time the control program run by the LANai processor becomes aware that an additional burst associated with a checkpoint operation of the j-th LP has been completed.

By the previous description, the ratio $\frac{\texttt{completed_transfers[j]}}{\texttt{total_transfers[j]}}$ keeps track of the advancement percentage of the last activated checkpoint operation, if any, for the j-th LP.

To determine whether additional bursts have been completed for a given checkpoint operation (this is required for updating the `completed_transfers[]` vector), the LANai control program keeps the index of the earliest not yet completed checkpoint operation packed in

checkpoint_slots[], and checks within its main loop whether terminal bits of the DMA control blocks in that entry have been set since the last check. In the positive case, the control program becomes aware that additional bursts of that checkpoint operation have been completed by the EBUS DMA for the LP whose identifier is maintained into the corresponding entry of LP_ids[]. As a consequence, the control program increments the counter maintained in the entry of completed_transfers[] associated with that LP by the amount of additionally completed bursts. In case the checkpoint operation is completed, the index of the earliest not yet completed checkpoint operation packed in checkpoint_slots[] is moved.

Upon re-synchronization for the j-th LP, the function ckpt_cond_abort() checks whether the condition $\frac{\texttt{completed_transfers[j]}}{\texttt{total_transfers[j]}} < \texttt{threshold}$ is satisfied. In the positive instance, the last activated checkpoint operation for that LP is aborted. This is implemented by making the LANai control program set to zero the field len in all the not yet executed DMA control blocks maintained in the checkpoint_slots[] entry associated with that operation. This has the effect to let the DMA controller pass through these blocks (with no DMA activation) without switching off the DMA channel 2. To efficiently identify the entry of checkpoint_slots[] maintaining information on the checkpoint operation that must be aborted we have used an additional vector of integers, namely indexing[]. The j-th entry of this vector keeps track of the index of the entry of checkpoint_slots[] associated with the last activated checkpoint operation of the j-th LP, if any. Also, in case a commit decision is taken for a not yet completed checkpoint operation of the j-th LP, the re-synchronization function ckpt_cond_abort() takes care of completing the operation through a memcpy() that copies the not yet checkpointed portion of the LP state vector into the checkpoint buffer.

Discussion on v2.4 vs v3.0. To track the advancement of a checkpoint operation the LANai control program is charged with similar amount of work in both CCL v2.4 and v3.0. Specifically, in CCL v2.4 the control program polls within its main loop the EBUS DMA to determine whether the last activated burst is completed (as discussed in Section 3.2 such a polling is also used to detect when to activate a new EBUS DMA operation). Instead, in CCL v3.0 the control program polls the terminal bits in the DMA control blocks maintained by the checkpoint_slots[] entries.

However, there is an issue related to the difference in the tracking mechanism of the advancement of a checkpoint operation that needs additional discussion. As already pointed out, in CCL v2.4, the LANai control program is not allowed to activate a new burst associated with the checkpoint operation unless the last activated one is already completed. This means that the counter completed_transfer, updated by the control program each time it discovers that the last activated burst has been completed, (possibly) underestimates the real advancement

state of the non-blocking checkpoint operation by at most a single burst. In case of CCL v3.0, all the bursts associated with a checkpoint operation are packed within the LANai memory (i.e. within the `checkpoint_slots[]` vector), and are analyzed/activated by the external controller of the EBUS DMA. According to this scheme, more than one burst might be carried out before the polling mechanism (performed by the LANai control program) checks the terminal bit within DMA control blocks and updates the counter within `completed_transfers[]`. As a consequence, this counter value, read by the re-synchronization function `ckpt_cond_abort()` run at the host side, might theoretically underestimate the real advancement percentage of the checkpoint operation by more than a single burst. This might have the following two effects:

- Underestimation by several bursts might impact the checkpoint commit/abort decision taken upon re-synchronization in case the advancement percentage of the operation is just around the threshold value we pass to the re-synchronization function. Specifically, the counter within an entry of `completed_transfers[]` might erroneously indicate that the checkpoint operation advancement is under the threshold while the real advancement is above the threshold.
- Underestimation by several bursts might lead to the situation in which, upon re-synchronization with a commit indication for the on-going non-blocking checkpoint operation, we might charge on the CPU non-minimal checkpointing work (i.e. data transfer between state buffer and checkpoint buffer) that has been already carried out by the EBUS DMA (but not yet detected as carried out since the corresponding counter within the `completed_transfers[]` vector still needs to be updated by the LANai control program by several units).

To test whether underestimation by several bursts is likely to occur in practice when using CCL v3.0, we have carried out a set of experiments demonstrating that polling performed by the LANai control typically keeps the information maintained by the `completed_transfers[]` vector quite updated. For the experiments we have used an architecture with Pentium III 866 MHz CPU, 32-bit/33-MHz PCI bus and M3M-PCI64C myrinet device running LINUX, namely the same architecture adopted for the experimental study on the PCS simulation application reported in Section 5.

We have implemented a test-bed application level software which cyclically issues a non-blocking checkpoint operation through the `non_block_ckpt()` function and then immediately invokes the re-synchronization function using a threshold value of zero. In other words, we issue a checkpoint operation and then immediately wait for its completion (i.e. no abort is

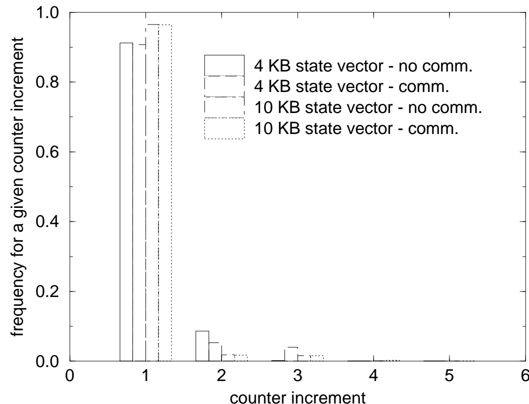

Figure 4: Histogram for the Counter Update.

ever forced with a threshold value set to zero). Also, the operation is entirely carried out through the EBUS DMA with burst length set to 1 KB as suggested by [21], i.e. we have used a re-synchronization software resulting as a modification of `ckpt_cond_abort()`, which does not implement the optimization of letting the checkpoint operation to be completed by the CPU in order to exploit the higher transfer rate of the system bus. This is done to better observe the relation between the real advancement of the checkpoint operation carried out through the EBUS DMA and the consequent updating of counter values maintained by the vector `completed_transfers[]`. We have repeated this cycle 10^6 times. The size of the state vector has been treated as an independent parameter in our study. Specifically, we have considered two cases, namely 4 KB and 10 KB state vector size.

Each time the LANai control program updates the information keeping track of the advancement state of the checkpoint operation, i.e. the counter within the vector `completed_transfers[]`, we have detected the amount of the counter increment. (Recall this happens thanks to polling on the chain of control blocks performed by the LANai control program.) This parameter provides indications on the delay the LANai control program experiences in updating the information as respect to the real advancement of the checkpoint operation. Specifically, the larger the counter increment, the longer the delay in the updating procedure, therefore, the larger the underestimation while tracking checkpoint advancement through polling on the chain of DMA control blocks.

We have tracked the counter increment under both the case of no communication and the case of active communication. In the latter scenario, we continuously send 64 bytes messages from a remote machine and locally extract these messages from the receive queue through a dedicated thread. Actually, the case of standing communication is unfavorable since it might

additionally delay the counter update due to delay in the execution of the polling mechanism on terminal bits within DMA control blocks at the level of the LANai control program. This is because the DMA engines require access to the LANai bus for performing communication tasks and the LANai processor has lower access priority to the LANai bus as compared to the DMA engines. Therefore the LANai control program might be relevantly delayed when these engines continuously require access to the LANai bus.

The measures, reported in the form of histogram in Figure 4, show that under the case of both standing and not standing communication, and independently of the state vector size, with frequency on the order of more than 0.9 the counter increment is of a single unit. This indicates that CCL v3.0 is able to maintain the information keeping track of the advancement state of a checkpoint operation almost as updated as in CCL v2.4. Specifically, underestimation of the advancement percentage of the checkpoint operation is expected to be of at most a single burst, just like the case of CCL v2.4.

As a final observation on the difference between the two implementations, we note that aborting a checkpoint operation in CCL v3.0 is slightly more complex than in CCL v2.4 since it requires the LANai control program to set to zero the `len` field in all the not yet executed DMA control blocks maintained in the entry of `checkpoint_slots[]` associated with that operation. This kind of work is not required by CCL v2.4 since aborting a checkpoint operation is actually implemented by preventing the control program to further activate the EBUS DMA for additional bursts associated with that checkpoint operation, i.e. by discarding work from the control program run by the LANai processor. However, the definitely higher speed of the LANai 9 processor is expected to still allow responsiveness of the control program in CCL v3.0 in spite of the slightly heavier work in case of checkpoint abort.

3.4 A Simulation Engine Exploiting CCL v3.0

Figure 5 shows the high level structure of an optimistic simulation engine, as described in [19], exploiting checkpointing capabilities offered by CCL v2.4. (For sake of simplicity, Global-Virtual-Time calculation and Fossil-Collection are omitted). A first re-synchronization point must be located just after the LP scheduling operation (see line 4). This is done in order to ensure data consistency by avoiding that the LP scheduled for event execution, namely *sched_LP*, issues updates on its state vector while the vector is still being transferred into the stack of checkpoints by the myrinet device (otherwise those updates might ultimately result in a non-consistent checkpoint for that LP). Re-synchronization in this point of the engine must be activated only in case *sched_LP* is associated with the last issued non-blocking checkpoint operation.

A second re-synchronization point must be located just before activating a checkpoint operation. This is due to the monoprogrammed checkpointing approach which does not allow issuing a new checkpoint request in case the last issued one is still being handled. Re-synchronization in this point must be activated only in case the test in line 9 results true, i.e. in case a checkpoint is required for *sched_LP*.

```
1  while(not end)
2    <receive messages/antimessages and update event lists>;
3    sched_LP = <schedule next LP>;
4    if (sched_LP lastly issued a non-block. ckpt request) [ckpt_cond_abort(threshold)];
5    if (rollback required for sched_LP)
6          <execute rollback for sched_LP>;
7    <event execution for sched_LP>;
8    <send event notification messages>;
9    if (checkpoint required for sched_LP)
10         [ckpt_cond_abort(threshold)];
11         non_block_ckpt(sched_LP, simulation_clock);
```

Figure 5: An Engine Suited for CCL v2.4.

The multiprogrammed checkpointing approach implemented by CCL v3.0 completely avoids re-synchronization in line 10 since any LP is allowed to issue a checkpoint request independently of the checkpointing work currently charged on the myrinet device. The outcoming engine exploiting CCL v3.0 is shown in Figure 6. In this engine, re-synchronization takes place in line 4 selectively for *sched_LP*, as allowed by the version of the re-synchronization function `ckpt_cond_abort()` associated with the API of CCL v3.0.

4 Related Work

For optimistic PDES systems, traditional approaches to reduce the checkpointing cost are based on (i) logging the LP state vector after an interval of executed events (see, e.g., [6, 18, 23, 27]) (instead of at each event), or (ii) employing an incremental checkpointing approach (see, e.g., [2, 24, 26, 29, 30]), or (iii) a combination of the two techniques (see, e.g., [7, 28]). All these solutions address the checkpointing cost by reducing the amount of data copied per simulation event. Compared to CCL, these solutions act at a higher abstraction level, since they are not tailored to reduce the per-byte CPU latency of the checkpoint operation. This is instead the target to the CCL project, which exploits DMA devices to perform the copy operation.

An approach closer in nature to CCL can be found in [9], where a hardware-based solution

```
1  while(not end)
2     <receive messages/antimessages and update event lists>;
3     sched_LP = <schedule next LP>;
4     ckpt_cond_abort(sched_LP, threshold);
5     if (rollback required for sched_LP)
6          <execute rollback for sched_LP>;
7     <event execution for sched_LP>;
8     <send event notification messages>;
9     if (checkpoint required for sched_LP)
10         non_block_ckpt(sched_LP, simulation_clock);
```

Figure 6: An Engine Suited for CCL v3.0.

to handle state log is provided, namely the rollback chip. The difference with our proposal is that this solution relies on specialized hardware, while CCL can be employed with COTS platforms.

In the context of fault tolerance, many solutions for checkpointing have been proposed (see, e.g., [5, 12, 17]), mostly aimed at optimizing memory-to-disk data transfer (disk transfer is required to support persistence of the state log in the presence of failures). Conversely, our proposal addresses the orthogonal issue of optimizing memory-to-memory data transfers for LP state logs, which need to be executed relatively frequently since they cope with the endemic phenomenon of rollback occurrence due to intra LP synchonization requirements in optimistic PDES.

5 An Experimental Study

In this section we report the results of a comparative empirical study of monoprogrammed and multiprogrammed non-blocking checkpointing, carried out using a PCS simulation application as a test-bed. Previous experimental studies have already shown the performance benefits from non-blocking checkpointing as compared to classical CPU charged (synchronous) checkpointing (in [19] those benefits have been evaluated exactly for the case of a PCS simulation application with simulation model features similar to those considered in this study). Therefore, we focus on a pure comparison between monoprogrammed and multiprogrammed non-blocking checkpointing. The experiments were all performed on a cluster of 4 Pentium III 866 MHz (256 Mbytes RAM). All the PCs of the cluster run LINUX (kernel version 2.2.15) .

Since CCL v2.4 has not been ported on M3M-PCI64C myrinet cards due to hardware incompatibility, a full legittimate comparison of CCL v3.0 and CCL v2.4 based on the same hardware cannot be performed. Hence, in order to achieve a fair comparison between mono-

programmed and multiprogrammed checkpointing by maintaining the same hardware platform for both the approaches, we have emulated the monoprogrammed approach through CCL v3.0. Specifically, the engine in Figure 5, has been emulated through the engine in Figure 6 by adding a call to the re-synchronization function proper of CCL v3.0 before any checkpoint request is issued by whichever LP, and passing the identifier of the LP that lastly activated a checkpoint request as the first parameter of the function. This avoids the exploitation of the batching mechanism offered by CCL v3.0 since at most a single checkpoint request is packed in the `checkpoint_slots[]` vector at any time. For completeness of the analysis, we have also performed a comparison between CCL v3.0 and CCL v2.4 running both versions on the same machines, but with different myrinet cards mounted on the machines (M3M-PCI64C for v3.0 vs M2M-PCI32C for v2.4). The results would provide indications of the benefits from the improved network level hardware/software technology underlying v3.0, compared to the technology underlying v2.4, while maintaining the same host-level hardware/software technology.

In the experiments, a non-blocking checkpoint operation is activated by the LP after the execution of each simulation event, thus obtaining a situation in which lack of checkpoints for particular LP state vector values is determined only on the basis of checkpoint abort decisions taken upon re-synchronization. As shown by experimental results in [19], this is a favorable test case for non-blocking checkpointing, allowing to fully exploit the effectiveness of an optimization in its implementation.

As a last preliminary observation, we have used a classical bound of 40 processed events [6] as the maximum distance between committed checkpoints in order not to incur performance problems due to interaction between Fossil-Collection and the frequency of Global-Virtual-Time calculation [6, 18, 23]. This is implemented by forcing checkpoint commit, i.e. by executing the `ckpt_cond_abort()` function with the parameter `threshold` set to 0, in case at least 40 events were executed by an LP since the last committed checkpoint of that same LP.

5.1 Simulation Model and Performance Metrics

In a PCS system, base stations provide communication services to mobile units. In our simulation model the service area is partitioned into cells, each modeled by a distinct LP. A cell represents a receiver/transmitter having either some fixed number of channels allocated to it (Fixed-Channel-Assignment, namely FCA) or a number of channels dynamically assigned to it (Dynamic-Channel-Assignment, namely DCA). We consider an FCA model. The model is call-initiated [4] since it only simulates the behavior of a mobile unit during conversation,

i.e. the movement of a mobile unit is not tracked unless the unit itself is in conversation. Therefore, the model is organized around two entities, namely cells and calls. Call requests arrive to each cell according to an exponential distribution [1, 3, 4] with inter-arrival time t_{int} seconds. All the calls initiated within a given cell are originated by the LP associated with that cell, therefore no external call generator is used.

The state vector of any LP records statistics, information about busy channels and, for each channel, information about features of the mobile unit involved in the ongoing call (e.g. scheduled call termination time, call initiation time, class of the mobile unit etc.), if any. As a result, the size of the state vector depends on the number of channels associated with the cell. We have selected a model with 50 channels per cell, which gives rise to LP state vector size of about 2 Kbytes.

There are three types of events, namely hand-off, due to mobile unit cell switch, call termination and call arrival. A call termination simply involves the release of the associated channel, whose identifier is maintained into the event compound structure, and statistics update. A call arrival checks if there is at least an available channel. In the negative instance, the incoming call is simply counted as a block, otherwise an available channel is allocated for the call and signal-to-interference-plus-noise ratio is computed in order to determine the power for the transmission on that channel [11]. The cost of this operation is dependent on the number of busy channels at the time of the call arrival. When a hand-off occurs between adjacent cells, the hand-off event at the cell left by the mobile unit simply involves the release of the channel and statistics update. Instead, the hand-off event at the destination cell checks for channel availability, and allocates an available channel, if any, for the call. If there is no available channel, then the call is simply cut off (dropped). Otherwise, signal-to-interference-plus-noise ratio calculation is required.

In our model there are two distinct classes of mobile units. Both of them are characterized by a residence time within a cell which follows an exponential distribution [1], with mean 3 minutes (fast movement units) and 30 minutes (slow movement units), respectively. The average holding time for each call associated with both fast and slow movement units is 2 minutes. When a call arrives at a cell, the type (slow or fast) of the mobile unit associated with the incoming call is selected from a uniform distribution, therefore any call is equally likely to be destined to a fast or a slow movement mobile unit.

Cells are modeled as hexagons, therefore all the cells, except bordering cells of the coverage area, have six neighbors. In the experimental study we have varied the number of LPs between 32 and 256, with even distribution of the LPs on the 4 machines of the cluster. Variation of the number of LPs allows us to compare monoprogrammed and multiprogrammed non-blocking

checkpointing while the rollback pattern of the parallel execution changes. Specifically, once fixed the size of the underlying computing platform, increasing the number of LPs leads to assign to each LP a reduced amount of CPU cycles per time unit. Hence, the divergence of their simulation clocks gets typically reduced, with a consequent reduction of the amount of rollback in the parallel execution.

We have used two different settings, characterized by different values for the expected call interarrival time per cell t_{int}, namely 3 seconds and 10 seconds. Given that the average call holding time is 2 minutes, we obtain for those interarrival times channel utilization factors of 80% and of about 25%, respectively. Actually, varying t_{int} produces a variation in the real granularity of simulation events. Specifically with a larger value for t_{int} the expected number of busy channels upon call arrival is kept low, therefore we get relatively low granularity for events that require channel allocation and signal to interference-plus-noise ratio calculation. On the other hand, we have longer expected execution time for these events in case of shorter values for t_{int} since we expect a non-minimal number of busy channels upon call arrival. Given that the number of channels per cell is the same in both settings, the LP state vector size does not vary. As a consequence, variation in the real granularity of simulation events allows us to compare monoprogrammed and multiprogrammed non-blocking checkpointing in case of different configurations with respect to the relation between the cost of event processing and the latency of a checkpoint operation.

We report measures related to the event rate, classically evaluated as the amount of committed simulation events per second, which is an indicator of the speed of the parallel execution when a given checkpointing approach is adopted. We report the peak event rate value observed while moving the value of the parameter `threshold`, passed to the `ckpt_cond_abort()` re-synchronization function, between 0.0 and 1.0, with step 0.1. We also report the values of the average distance between committed checkpoints observed in correspondence of the peak event rate value. This parameter indicates the ability of a given approach to carry out useful checkpointing work (i.e. to commit checkpoints) at the point in which the execution speed is maximized. The shorter the average distance between committed checkpoints, the higher such an ability. Note that the average distance between committed checkpoints is also an indicator of the cost of performing each single state recovery upon rollback (the larger the distance between committed checkpoints, the longer the expected state recovery cost due to coasting forward [6, 18, 23]). Therefore, in combination with the frequency of rollback, it is an indicator of the state recovery cost paid at the point in which the execution speed is maximized vs `threshold`. (Some data related to the frequency of rollback and the efficiency of the parallel execution will also be reported.) Each reported data point results as the average

of 10 runs all done with different random seeds.

5.2 Results

The results are plotted in Figure 7 and in Figure 8 for the case of $t_{int} = 10$ seconds and $t_{int} = 3$ seconds, respectively. By the event rate plots in Figure 7 we get that the multiprogrammed approach allows a relevant increase in the speed of the execution. Specifically, when the number of LPs is 32, multiprogrammed non-blocking checkpointing improves the execution speed of about 12% compared to the case of monoprogrammed checkpoiniting emulated via CCL v3.0 (this gain is further improved to 21% compared to the case of CCL v2.4 run on the less efficient M2M-PCI32C myrinet technology). The gain tends to decrease for larger amounts of LPs, however it remains in the order of about 6% even for the case of 256 LPs (compared to CCL v2.4 it remains at about 7%). By the plots related to the average distance between committed checkpoints we get the basic reason for this behavior. In particular, the plot for the average distance between committed checkpoints produced by the multiprogrammed approach is almost flat, with values in the order of about 1/1.5 events. In other words, at the point in which the execution speed is maximized vs `threshold` (recall the plotted points refer to the best observed event rate vs this parameter), the multiprogrammed approach allows commitment of the most part of the checkpoint requests, independently of the number of LPs. Instead, monoprogrammed checkpointing, either emulated via CCL v3.0 or supported via CCL v2.4, produces an average distance between committed checkpoints in the order of about 18/25 events, thus resulting in a reduced ability to carry out useful checkpointing work (i.e. committed checkpoints). This increases the cost of state recovery operations due to coasting forward [18], which, in its turn, negatively impacts the performance especially for larger values of the rollback frequency, which is typical of executions with a lower amount of LPs per machine. With respect to the latter assertion, we have observed that the rollback frequency is in the order of 4.0% when the number of LPs is 32 and then decreases to a minimum of about 1.5% when the number of LPs is 256, passing through 2.5% for intermediate values. Correspondingly, the efficiency of the parallel execution, i.e. the ratio between the number of committed events and the total number of executed events, ranges between 90% and 95%. These values are the same for monoprogrammed and multiprogrammed checkpointing.

The results in Figure 8 for shorter call interarrival time, namely $t_{int} = 3$ seconds, confirm the tendency observed in case of $t_{int} = 10$ seconds. The major difference is that the gain provided by the multiprogrammed approach is reduced. The reason for this behavior is in the strongly reduced distance between committed checkpoints produced by the monoprogrammed approach, especially for more than 32 LPs. Such a reduction comes out from the increase in

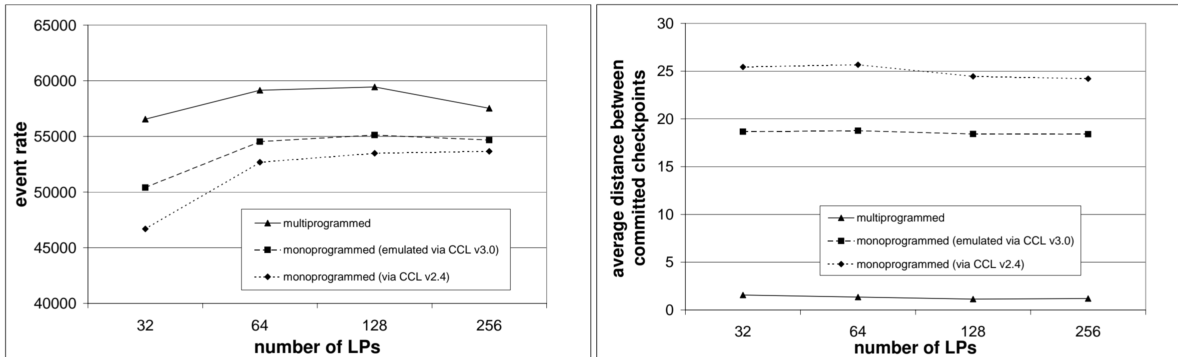

Figure 7: Results for Call Interarrival Time 10 seconds.

the granularity of some event types, which sometimes produces an increase in the expected length of the interval between the activation of a checkpoint operation (see line 11 of the engine in Figure 5) and the successive re-synchronization occurrence due to device contention (see line 10 of the same engine). (Recall that reduction of t_{int} increases the expected event granularity since we get an increase of the expected number of busy channels upon call arrival, therefore there is an increase in the cost of signal-to-interference-plus-noise ratio calculation.) Such an increase allows the monoprogrammed approach to exhibit better ability to carry out useful checkpointing work due to the reduction of the impact of re-synchronization on the abort of checkpoints.

An interesting additional observation is related to the fact that, for $t_{int} = 3$ seconds, the plot for the distance between committed checkpoints produced by the monoprogrammed approach does not stay flat, as instead happens for $t_{int} = 10$ seconds. This is because for frequent call arrivals, the expected event granularity increases vs the number of LPs due to the variation of the impact of bordering effects. Specifically, the percentage of cells adjacent to the border cells is larger for lower values of the number of LPs. Also, in case of frequent call arrivals, these cells are more likely to get saturated due to the handoffs moving mobiles from the bordering cells to these cells (recall $t_{int} = 3$ seconds leads to channel utilization factor of 80%) ([4]). As a consequence, a larger percentage of call arrival events find a saturated cell in case of lower number of LPs. This gives rise to a reduction of the real event granularity since no channel assignment takes place in case the cell is saturated upon call arrival, thus no signal-to-interference-plus-noise ratio calculation must be performed. The impact of border effects on the reduction of the real event granularity decreases vs the number of LPs, which is

[4]In our simulation model, mobiles are not allowed to get out of the coverage area.

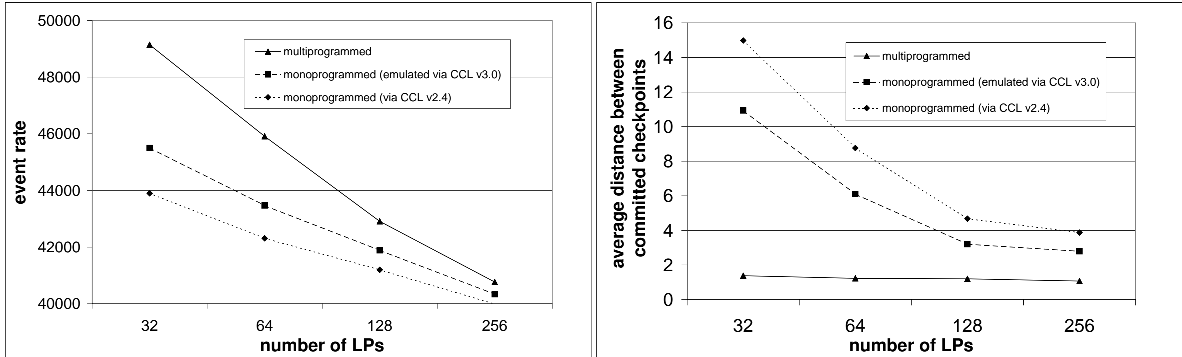

Figure 8: Results for Call Interarrival Time 3 seconds.

the reason why the monoprogrammed approach exhibits a lower distance between committed checkpoints for an increased number of LPs. This is also one reason for the decrease of the event rate vs the number of LPs independently of the adopted checkpointing approach (monoprogrammed vs multiprogrammed).

Anyway, for $t_{int} = 3$ seconds, the gain of the multiprogrammed approach for 32/64 LPs remains on the order of 8/6% compared to the monoprogrammed approach emulated via CCL v3.0, and on the order of 12/11% compared to the monoprogrammed approach via CCL v2.4. For completeness of the exposition, we mention that the rollback frequency and the efficiency values vs the number of LPs are similar to those observed in case of $t_{int} = 10$ seconds.

Overall, as soon as some event types have relatively large granularity, the monoprogrammed approach tends to provide performance similar to the multiprogrammed approach since there is higher likelihood that non-blocking checkpoint operations have already been completed before re-synchronization occurrence due to device contention (being themselves carried out concurrently with event execution). In other words, the monoprogrammed approach exhibits reduced impact of re-synchronization due to device contention on checkpoint aborts, and thus on performance. This is exactly the objective of the multiprogrammed approach that, by its nature, achieves such an impact reduction also for execution with fine event granularity, case in which re-synchronization due to device contention has real potential for degrading the execution speed when monoprogrammed non-blocking checkpoints are employed.

6 Summary

In previous implementations of non-blocking (CPU offloaded) checkpointing in support of optimistic discrete event simulation, re-synchronization not only was invoked to prevent data inconsistency, but also to cope with concurrent checkpoint requests on the same checkpointing device. This paper explores the multiprogrammed approach to perform non-blocking checkpoint operations, thus removing the need for re-synchronizations due to contention on the device (the checkpoint requests are batched and eventually served by the device). An implementation of the Checkpointing-and-Communication Library, namely CCL v3.0, capable to support the multiprogrammed execution mode on a myrinet-based cluster environment is presented, and the results of an experimental study on a Personal Communication System (PCS) simulation application, which confirm the reduction of the negative impact of re-synchronization on performance, are reported.

References

[1] A. Boukerche, S. K. Das, A. Fabbri, and O. Yildz. Exploiting model independence for parallel PCS network simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, pages 166–173. IEEE Computer Society, May 1999.

[2] D. Bruce. The treatment of state in optimistic systems. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 40–49. IEEE Computer Society, June 1995.

[3] C. D. Carothers, D. Bauer, and S. Pearce. ROSS: a high performance modular Time Warp system. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation*, pages 53–60. IEEE Computer Society, May 2000.

[4] C. D. Carothers, R. M. Fujimoto, and Y. B. Lin. A case study in simulating PCS networks using Time Warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 87–94. IEEE Computer Society, June 1995.

[5] E. Elnozahy, D. Johnson, and W. Zwaenepoel. The performance of consistent checkpointing. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 39–47. IEEE Computer Society, 1992.

[6] J. Fleischmann and P. Wilsey. Comparative analysis of periodic state saving techniques in time warp simulators. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 50–58. IEEE Computer Society, June 1995.

[7] S. Franks, F. Gomes, B. Unger, and J. Cleary. State saving for interactive optimistic simulation. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 72–79. IEEE Computer Society, June 1997.

[8] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[9] R. M. Fujimoto, J.-J. Tsai, and G. Gopalakrishnan. Design and evaluation of the rollback chip: Special purpose hardware for Time Warp. *IEEE Transactions on Computers*, 41(1):68–82, 1992.

[10] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and System*, 7(3):404–425, July 1985.

[11] S. Kandukuri and S. Boyd. Optimal power control in interference-limited fading wireless channels with outage-probability specifications. *IEEE Transactions on Wireless Communications*, 1(1):46–55, 2002.

[12] K. Li, J. Naughton, and J. Plank. Low latency concurrent checkpointing for parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):474–479, 1994.

[13] MYRICOM. Lanai 4. *Draft*, Feb. 1999.

[14] MYRICOM. Lanai 9. *http://www.myri.com/myrinet/performance/index.html*, May 2001.

[15] MYRICOM. PCI64 programmer's documentation. *http://www.myri.com*, May 2001.

[16] S. Pakin, M. Lauria, and A. Chen. High performance messaging on workstations: Illinois Fast Messages (FM) for Myrinet. In *Proc. of Supercomputing'95*. ACM/IEEE Computer Society, Dec. 1995.

[17] J. Plank, M. Beck, and G. Kingsley. Libckpt: Transparent checkpointing under UNIX. In *Proceedings of USENIX Winter Technical Conference*, pages 213–223. USENIX Association, 1995.

[18] B. R. Preiss, W. M. Loucks, and D. MacIntyre. Effects of the checkpoint interval on time and space in Time Warp. *ACM Transactions on Modeling and Computer Simulation*, 4(3):223–253, July 1994.

[19] F. Quaglia and A. Santoro. Non-blocking checkpointing for optimistic parallel simulation: Description and an implementation. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):593–610, 2003.

[20] F. Quaglia and A. Santoro. Modeling and optimization of non-blocking checkpointing for optimistic parallel simulation on myrinet clusters. *Journal of Parallel and Distributed Computing*, 65(6):667–677, 2005.

[21] F. Quaglia, A. Santoro, and B. Ciciani. Tuning of the checkpointing and communication library for optimistic simulation on myrinet based NOWs. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 241–248. IEEE Computer Society, 2001.

[22] F. Quaglia, A. Santoro, and B. Ciciani. Conditional checkpoint abort: an alternative semantic for re-synchronization in CCL. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation*, pages 143–150. IEEE Computer Society, May 2002.

[23] R. Ronngren and R. Ayani. Adaptive checkpointing in Time Warp. In *Proc. of the 8th Workshop on Parallel and Distributed Simulation*, pages 110–117. Society for Computer Simulation, July 1994.

[24] R. Ronngren, M. Liljenstam, R. Ayani, and J. Montagnat. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 70–77. IEEE Computer Society, May 1996.

[25] A. Santoro and F. Quaglia. PCI-DMA/CPU handoff for increased effectiveness of checkpointing functionalities in CCL. In *Proceedings of the 7th International Symposium on Distributed Simulation and Real-Time Applications*, pages 120–127. IEEE Computer Society, 2003.

[26] A. Santoro and F. Quaglia. Transparent state management for optimistic synchronization in the High Level Architecture. *SIMULATION: Transactions of The Society for Modeling and Simulation International*, 82(1):5–20, 2006.

[27] S. Skold and R. Ronngren. Event sensitive state saving in Time Warp parallel discrete event simulation. In *Proc. of the Winter Simulation Conference*, pages 653–660. Society for Computer Simulation, 1996.

[28] H. Soliman and A. Elmaghraby. An analytical model for hybrid checkpointing in Time Warp distributed simulation. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):947–951, 1998.

[29] J. Steinman. Incremental state saving in SPEEDES using C plus plus. In *Proceedings of the Winter Simulation Conference*, pages 687–696. Society for Computer Simulation, 1993.

[30] D. West and K. Panesar. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 78–85. IEEE Computer Society, May 1996.