View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by St Andrews Research Repository

On Page-based Optimistic Process Checkpointing

Alan Dearle

Department of Computing Science
University of Stirling,
Stirling, FK9 4LA, Scotland
al@cs.stir.ac.uk

David Hulse

Department of Computer Science
University of Adelaide,
S.A., 5005, Australia
dave@cs.adelaide.edu.au

Abstract

Persistent object systems must provide some form of checkpointing to ensure that changes to persistent data are secured on non-volatile storage. When processes share or exchange modified data, mechanisms must be provided to ensure that they may be consistently checkpointed. This may be performed eagerly by synchronously checkpointing all dependent data. Alternatively, optimistic techniques may be used where processes are individually checkpointed and globally consistent states are found asynchronously. This paper examines two eager checkpointing techniques and describes a new optimistic technique. The technique is applicable in systems such as SASOS, where the notion of process and address space are decoupled.

1. Introduction

Persistent systems provide programmers with a uniform view of volatile and non-volatile memory. The benefits of orthogonal persistence have been expounded elsewhere and we shall not labour them here. All persistent systems must be capable of establishing self-consistent recoverable states on some non-volatile medium which may be used to restart an arbitrary computation following an orderly shutdown or failure. The process of forming such states has been variously called checkpointing, snapshotting and stabilisation.

Forming a checkpoint involves securing changes on a stable medium. Commonly this involves writing a change log and/or flushing those pages modified since the last checkpoint onto disk; in this paper we shall focus on the page based approach. New checkpoints must be established with care, as they must not endanger the integrity of the previous checkpoint. In many persistent systems this is achieved using a *shadow paging* technique [13] combined with an atomic commit operation [1]. When shadow paging is employed, a dirty page never overwrites a clean version of a page in the stable store. Instead, dirty pages are written to another location (the shadow), with these copies of the pages being known as *shadow copies*. When a page is modified for the first time, a shadow location for it is allocated on non-volatile storage. In many systems at most a single shadow copy of each page exists and once a shadow copy of a page exists, that page is used for future operations. Such stores are known as *bistable* stores.

The establishment of a checkpoint changes the status of shadow pages so that they become the clean versions and further modification of the pages results in the allocation of new shadow pages whilst the old clean pages are freed for reuse. As the original pages are overwritten only after successful completion of the atomic checkpoint operation, the shadow copies permit the system to roll back should a failure occur.

In distributed systems in which the state of one node can become dependent on data stored on another node, some mechanism must be provided to ensure that individual checkpoints form a globally consistent state. The need to ensure that process checkpoints are consistent also arises in single-node systems which provide shared memory or inter-process communication. The simplest approach to global consistency is to enforce system wide simultaneous checkpoint in which the kernel(s) block all executing processes and copy modified data to some stable medium. This approach has been implemented in the Monads system [6]. The disadvantages of this approach are: the entire system freezes whilst the stabilise occurs, and the technique does not scale well. There are three basic alternatives to this approach:

1. Prevent processes from exchanging and sharing data modified with respect to the stable store: this allows processes to be individually checkpointed.
2. Determine which processes and data are causally dependent upon each other's state and only force these to stabilise together, leaving the rest of the system to run.
3. Permit optimistic process checkpoint, in which processes are individually checkpointed, and globally consistent states are found asynchronously.

The first two approaches automatically result in globally consistent states. However, the third approach requires some policy, such as eager checkpointing, to ensure that globally consistent states may always be found. The use of copy-on-write techniques to protect RAM resident pages from modification may be used to optimise any of these techniques, sometimes with great effect [4]. Wu and Fuchs [18] describe a hardware implementation of the first approach in which processes are prevented from sharing modified data. This is achieved by forcing processes to perform checkpoint operations as soon as another client

requests the use of any updated data. A major concern of their work has been to limit roll-back propagation, so that the failure of any client affects only that client. This paper examines two alternative implementations of the second approach and introduces a new technique based on optimistic process checkpointing.

2. Causality and Causal Histories

The notion of the causal history of a process is central to process checkpointing. Causality can intuitively be defined in terms of some entity being in some way dependent upon the state of another. Formally, causality may be defined in terms of the *happened-before* relation [10]. Briefly, the happened before relation "$\rightarrow$" is the smallest relation that satisfies the following criteria:

- If a and b are events in the history of the same entity and a comes before b then $a \rightarrow b$
- If a is the sending of information from one entity and b is its receipt in another then $a \rightarrow b$
- If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$
- If $a \not\rightarrow b$ and $b \not\rightarrow a$ then a and b are said to be concurrent.

One approach to the analysis of causality is via time diagrams consisting of a series of horizontal lines with each line representing an entity with time increasing to the right. Communication between processes is represented by a directed arc between time lines. For example, consider the example of readers and writers: a process *P1* modifies an object *O1* and another process *P2* reads the modified object. This sequence of events is represented by the time diagram shown in Figure 1.

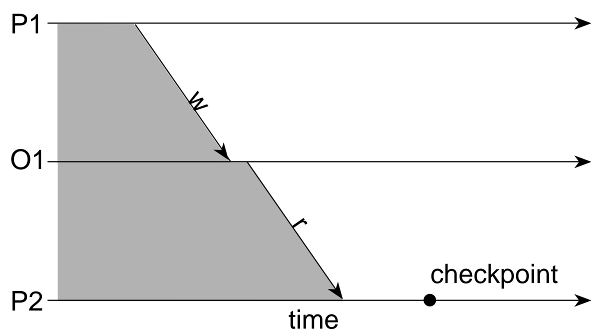

Figure 1: A time line diagram

Synchronous checkpointing techniques eagerly calculate the events that *happened before* a checkpoint request from a process and secure those events on stable storage. If we consider the checkpointing of process *P2*, the checkpoint must include the shaded region to yield a globally consistent state. Formally, such a state is termed a *consistent cut* which is a subset of the events which comprise the system such that, if e is an element of the consistent cut and $e' \rightarrow e$ then e' is also an element of the cut [14].

Graphically, consistent cuts are easily conveyed as a line drawn downwards through a time diagram partitioning the diagram into two parts with the past on the left and the future on the right. A cut is consistent if no arc (a communication) starts in the future and ends in the past. The intersection of the cut line and a process' time line represents the time at which a checkpoint was made. Checkpointing of individual processes and objects may be performed independently provided that consistent cuts can later be found.

It has been shown by Schwarz and Mattern [16] that *vector time* [5] may be used to characterise consistent cuts. In systems that employ vector time, each entity maintains a time vector, in which the elements represent knowledge about other entities within the system. Vector time is maintained as follows:

- Each vector conceptually has as many elements as there are entities within the system. $VT_i[j]$ denotes the jth element of the time vector for entity i.
- Each element of every vector is initially zero.
- On each atomic action (including message receipt and transmission), the entity increments the element of its vector corresponding to itself, that is $VT_i[i] \leftarrow VT_i[i] + 1$.
- Whenever a message is sent to another entity, the sender's vector is transmitted with the message.
- Upon receipt of a message, the receiving entity updates its own vector by constructing the piecewise maximum of its own and the sender's vector.

Johnson and Zwaenepoel [9] describe a technique for finding consistent cuts which involves the construction of a *dependency matrix* whose rows correspond to the vector times of all entities in the system. They show that a dependency matrix M represents a consistent cut iff,

$$\forall i,j \; Mij \leq Mjj.$$

In other words, the elements in column j must be less than or equal to the jth element. This intuitively states that no entity depends upon an event that happened after the consistent cut was established.

Whilst the algorithm of Johnson and Zwaenepoel is centralised, Johnson has developed a similar algorithm that may be applied in a distributed environment [8].

3. Synchronous Checkpointing

In this section we describe two synchronous approaches to checkpoint formation. The systems examined are the CASPER system [17] and a system developed by Jalili and Henskens [7] used in the Monads system [15]. Each of these systems eagerly finds consistent cuts by determining which processes and data are causally dependent upon each other's state, and forces these to checkpoint synchronously, leaving the other processes and data unaffected.

3.1 CASPER

The CASPER system [17] supports persistent distributed shared memory over a set of client processes which run on individual workstations. In CASPER, only those subsets of clients that share modified data must stabilise together. Such

interdependent clients are termed *associates* and a set of mutually dependent clients an *association*. Each client always belongs to exactly one association. Associations are dynamic in nature and are constructed between checkpoints by the server. Associations may merge over time due to the sharing of data between previously independent associations.

In CASPER, associations are maintained by a centralised stable store server. Each association has a corresponding page list, which identifies those pages modified or accessed by members of the association since their previous checkpoint; this information is used to incrementally build the associations. Conventional virtual memory techniques are used to detect reads and writes to pages. When a page is first modified, the identity of the modified page is added to the page list of the association containing the client performing the modification. Whenever a modified page is first accessed by a client, that client is added to the association containing the modified page. Since clients always belong to exactly one association, this may involve two associations and their associated page lists merging.

When a checkpoint is initiated by a client, only those clients belonging to the initiating client's association need be included. All modified pages held by checkpointing clients must be returned to the stable store server where they are written back to the store as an atomic operation. At the end of the checkpoint, the stable store will have moved into a new, consistent state. The association's page list can be used to determine which store pages are to be returned to a free page list, since up-to-date copies of those pages are stable and old versions are no longer useful. Since the associates no longer share modified pages after a checkpoint, the association concerned separates into sets each containing one client.

The use of associations in CASPER reduces the amount of synchronous checkpointing required. However, as identified by Jalili and Henskens, they force more data to be checkpointed than is strictly necessary. Consider the case shown in Figure 1 where process *P1* modifies an object that is later read by process *P2*. In CASPER processes *P1* and *P2* are placed in the same association meaning that if process *P1* is checkpointed so is process *P2* and vice-versa. As Jalili and Henskens point out, if process *P2* checkpoints so must process *P1* since the modification event is in the causal history of *P2* and is volatile. However, if process *P1* checkpoints, process *P2* does not have to since P1 is not dependent on any events in the causal history of P2 which has only read the data. This observation has lead to a refinement of the associations concept based on the maintenance of *directed dependency graphs*.

3.2 Directed Dependency Graphs

Like the CASPER system, the system described by Jalili and Henskens [7] tracks page reads and writes. The information gathered is used to construct directed dependency graphs (DDGs) rather than simple associations. A process may perform one of three operations on a page:

1. It may read from an unmodified page (with respect to the last checkpoint), which results in no dependency.
2. It may read a modified page which results in a directed dependency from the reading process to the modified page.
3. It may modify a page which results in a mutual dependency between the modifying process and the modified page.

As an example of this system, consider again the example of readers and writers: a process *P1* modifies a virtual page *V1* resulting in a mutual dependency between *P1* and the page. Another process *P2* reads the modified page making a uni-directional dependency between *P2* and *V1*. The resulting DDG is shown in Figure 2.

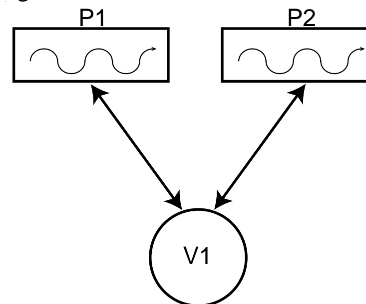

Figure 2: A Directed Dependency Graph representing the events shown in Figure 1.

To support the construction of directed dependency graphs the operating system kernel must maintain information describing:

1. which pages have been modified since the last checkpoint,
2. which pages have been accessed by the executing process, and
3. which pages have been modified by the executing process.

When a process checkpoints, the kernel traverses the DDG starting from the requesting process and transitively checkpoints all entities reachable via outgoing arcs. Thus, in the above example, if process *P1* checkpoints the state of *P1* and page *V1* will be checkpointed. If process *P2* checkpoints *P1*, *P2* and *V1* will be checkpointed. Like the CASPER system, this traversal eagerly forms consistent cuts. The use of DDGs rather than associations has increased the accuracy of the causal dependency information. However, the increased accuracy required to build dependency graphs is not without cost. Consider the following sequence of events:

1. Some page *V1* is read by process *P1*,
2. A context switch occurs,
3. V1 is modified by another process *P2*,
4. A context switch occurs,
5. Process *P1* reads the modified page *V1*.

This sequence is represented by the time diagram shown in Figure 3. When a process reads a virtual page it can continue to read that page without any page faults occurring until the virtual page is invalidated (removed from the MMU and TLB cache).

Consequently, other than the first read or write to a page, the exact ordering of events cannot be characterised using conventional virtual memory alone. Consider the example shown in Figure 3, the first read of page *V1* by process *P1* accesses an unmodified page. However, since that page is in the TLB of *P1*, it can read modifications by other processes, such as those made by process *P2*, without detection. Some additional mechanisms are therefore required if directed dependency graphs are to reflect the correct ordering of events.

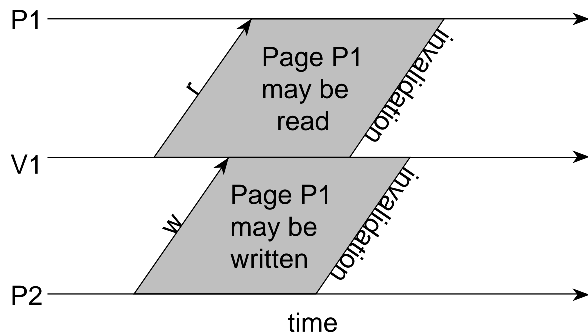

Figure 3: Inexact read/write events

Jalili and Henskens achieve this by recording those pages accessed during a time quantum of a process' execution and updating the dependency graphs lazily when the process is descheduled. In order to avoid the problem illustrated above, all context information is discarded on every context switch (the MMU is flushed and TLB cache thrown away).

The aim of the directed dependency graph approach is to reduce the false causality inherent with associations. Whilst the Jalili and Henskens approach largely achieves this aim, a small amount of false causality may still occur when swapping occurs. To illustrate this, consider a virtual page V1 which has been modified by process P1 resulting in a bi-directional arc between V1 and P1. Later this page is swapped out and never accessed by P1 again. The modification of V1 by another process P2 results in another bi-directional arc between P2. If P1 is checkpointed the system will cause P2 to be checkpointed even though P1 is not causally dependent on P2 since it has not seen the update of V1 by P2.

It should be clear that this technique reduces the amount of synchronous checkpointing that is required. Another important observation is that the use of either associations or directed dependency graphs guarantees that a consistent cut is always established following a checkpoint operation. An alternative approach to synchronous checkpointing is to checkpoint processes and their data independently and construct globally consistent cuts lazily.

4. Optimistic Checkpointing

This section describes a new technique to support optimistic process checkpointing. The technique extends the work of Jalili and Henskens and draws on the work of Johnson and Zwaenepoel. Rather than eagerly checkpointing all causally dependent processes, the technique uses optimistic checkpointing where each process may be checkpointed separately requiring consistent cuts to be found lazily. Each individual checkpoint has an associated vector time which permits consistent cuts to be found optimistically. The manner in which this is performed is beyond the scope of this paper, however the techniques described by Johnson and Zwaenepoel [9] would satisfy this requirement.

Jalili and Henskens use virtual page invalidation to impose an ordering on events to ensure a high degree of accuracy in causal tracking. In contrast, our technique tolerates an inexact ordering of events to improve performance. This is achieved by invalidating the MMU and TLB caches only at the time of checkpoint and page eviction. This approach introduces the potential for false causality which may be reduced by eagerly invalidating and checkpointing pages.

The system maintains two data structures to track accesses to physical memory called the Physical Readers List (PRL) and the Physical Writers List (PWL). Each of these is conceptually implemented as an array of lists of process identifiers where each array element corresponds to a page frame of physical memory. Whenever a process first reads from a frame of physical memory the identity of the process is added to the PRL and on the first write its identity is added to the PWL. Tracking accesses to physical rather than virtual memory, the PRL and PWL data structures are smaller and hence more manageable.

The PRL and PWL collectively comprise the DDGs constructed by Jalili and Henskens. On checkpoint, rather than traversing the closure of directed arcs in the DDG, our technique only traverses the arcs leading to modified pages accessed by the checkpointing process. Thus the checkpoint may be dependent on the state of other un-checkpointed processes. This dependency information is captured in the PRL and PWL and, since the PRL and PWL are dynamic entities, must be independently stored with the checkpoint so that a globally consistent state may later be formed. Vector times are therefore associated with each checkpoint to capture this dependency information. In the algorithms shown below, the vector time is lazily constructed at checkpoint and captures all the inter-process communications that have happened since the last checkpoint. The use of vector times has the additional benefit that they may be utilised in a distributed environment.

Each process maintains a logical clock called *localTime* which is incremented on read and write faults and when a checkpoint occurs. In addition to this scalar time, each process p maintains a vector time VT_p. This vector is updated lazily on checkpoint and page invalidation and represents the dependencies that the process has on other processes. This vector is only guaranteed to have the correct time on checkpoint.

Tracking accesses to physical rather than virtual memory introduces an additional problem, that of swapping. The PRL and PWL associated with each physical page effectively record part of the directed dependency graph. When a page is removed from physical memory (and hence the graph), the dependencies represented by the DDG must be maintained. This is achieved by eagerly updating the vector times of the processes which depend on it and associating this dependency information with the swapped page. A data structure called the Non-Resident Modified Page List (NRMPL) is therefore maintained which stores the identity of modified non-resident pages and a vector time to record accesses to those pages.

The NRMPL may have a single vector time associated with it; this produces false causality since passive pages located on disk cannot communicate but only requires the maintenance of a single vector time. Alternatively, a vector time may be maintained for each page in the NRMPL. This is expensive in terms of space but maintains causal accuracy and requires less computation to maintain causality information.

When a process is checkpointed, the modified pages that the process has accessed but are swapped out must become part of the checkpoint. For this reason a per process data structure called the Process Access List (PAL) is maintained to record the modified pages that have been swapped out and accessed by the process.

Our technique may be described by considering the five events that control its operation namely: read fault, write fault, page invalidation by reader, page invalidation by writer, and checkpoint.

4.1 Read Fault on page *va* at physical *pa* by process *p*:

When a read fault occurs, the process identity is added to the PRL associated with the physical page frame at which the virtual page is located. If the page is resident on swap space the page must have been modified since the last checkpoint. Consequently the reading of a page from swap may represent a communication of information from another process. The processes that have modified the page are encoded in the vector time stored in the NRMPL. For this reason the faulting process' vector time is updated when the page is read. We assume that a function called *VT_NRMPL* will recover this vector time. This is one of the few times that eager update of vector times takes place. Note that the page is left in the NRMPL until a writer of the corresponding physical page exists. This is necessary since only writers can propagate dependency information. As the process has performed an operation its logical clock is incremented – this advances time for the process. The read fault pseudo-code is shown in figure 4.

```
! If page has been swapped, update vt on process p
! from vt on swap
if va ∈ NRMPL do
    VT_p ← PIECEWISE_MAX(VT_p,VT_NRMPL(va))
PRL(pa) ← PRL(pa) + p
localTime( p ) ← localTime( p ) + 1
```

Figure 4: Handling read faults

4.2 Write Fault on page *va* at physical address *pa* by process *p*:

The code for a write fault is similar to that for a read fault and is shown in Figure 5. There are two main differences: first the identity of the faulting process is added to the PWL rather than the PRL for obvious reasons. Secondly, if the page has been brought into memory from swap space, it is removed from the NRMPL. This is possible since, after vector time update, the faulting process holds all causality information associated with the page. Furthermore since it is a writer, it will pass that information on should the page be invalidated or checkpointed.

```
! If page has been swapped, update vt on process p
! from vt on swap
if va ∈ NRMPL do
{
    VT_p ← PIECEWISE_MAX( VT_p, VT_NRMPL(va) )
    ! remove va once a writer has a copy.
    NRMPL ← NRMPL – va
}
PWL(pa) ← PWL(pa) + p
localTime( p ) ← localTime( p ) + 1
```

Figure 5: Handling write faults

4.3 Invalidate virtual page *va* at physical address *pa* by reader *p*:

The code shown in Figure 6 is executed when access to a physical page is taken away from a process that is reading that page. Commonly this will occur when a (virtual) page is swapped out by the kernel. However the page may or may not remain resident after invalidation. The code performs three functions: most importantly, it ensures that any communications from writers of the page being invalidated are recorded in the vector time of the invalidating process. The vector time of the invalidating process is updated using the *COMMUNICATE* procedure shown in Figure 6. This procedure updates the vector time of the invalidating reader in a non-transitive fashion. The second function of the code is to add the page to the PAL if it is modified; this records the fact that the process has accessed that modified page. Lastly, it removes the invalidating process from the PRL.

```
COMMUNICATE( sender, receiver : process )
{
    localTime( receiver ) ← localTime( receiver ) + 1
    VTreceiver( receiver ) ← localTime( receiver )
    VTreceiver ( sender ) ←  localTime( sender )
}
```

```
if modified(pa) do
{
    ! Lazily update VT of process p
    ∀ P in PWL(pa), P ≠ p, COMMUNICATE( P,p )
    ! Remember that process has accessed this page
    PAL(p) ← PAL(p) + va
}
! Update physical readers list
PRL(pa) ← PRL(pa) – p
```

Figure 6: Invalidating pages by reader

```
! Record all communications associated with
! this process and page
∀ P in PRL(pa) ∪ PWL(pa), P ≠ p,
        COMMUNICATE( p,P )
! Remember that the process has accessed this page
PAL(p) ← PAL(p) + va
! Update physical readers/writers list
PRL(pa) ← PRL(pa) – p
if p ∈ PWL(pa) do
{
    PWL(pa) ← PWL(pa) – p
    ! Update NRMPL if last extant writer
    if PWL(pa) = ϕ do
    {
        NRMPL ← NRMPL + va
        ! Record causally dependent processes
        VT_NRMPL( va ) ← VT_p
    }
}
```

Figure 7: Invalidating pages by writer

4.4 Invalidate virtual page *va* at physical address *pa* by writer *p*:

The code shown in Figure 7 is executed when access to a physical page is removed from a process that has written to that page. The code is based on the invalidating readers code but is slightly more complex. Firstly, the fact that process *p* potentially sent information to all processes that have read or written to the page at physical address *pa* must be recorded. This is performed in the first line of the algorithm. Next the page must be added to the PAL to record the fact that it should be included in any checkpoint of that process. Next the PRL and PWL are updated; the readers list needs to be included here since the process may have originally read the page. The last part of the code is activated if the writer is the last extant writer of the page. If it is, the causal history embodied in that page must be recorded in the NRMPL. Thus, if any future reader or writer faults on the page, the correct history of writers will be transferred to the new process.

```
! 1: Make the vt reflect (non-transitive) dependencies.
localTime( p ) := localTime( p ) + 1
for each pa in RAM: p ∈ ( PRL(pa) ∪ PWL(pa) )
    for each P ∈ PWL(pa), P ≠ p,
        COMMUNICATE( P,p )
! 2: Checkpoint pages
checkpoint all pages in:
    { pa: PWL(pa) ≠ ϕ and
        p ∈ ( PRL(pa) ∪ PWL(pa) ) } ∪ PAL(p)
! 3: Set VT of checkpoint
    VT_checkpoint ← VT_p
! 4: Reset PAL
    PAL(p) ← ϕ
! 5: Invalidate the pages that have been checkpointed.
for each page pa: p ∈ ( PRL(pa) ∪ PWL(pa) )
{
    ! Record all communications associated with
    ! this process and page
    if p ∈ PWL(pa)
        ∀ P in PRL(pa) ∪ PWL(pa), P ≠ p,
                COMMUNICATE( p,P )
    Invalidate TLB entry of page pa
    PRL(pa) ← PRL(pa) – p
    if p ∈ PWL(pa) do
    {
        PWL(pa) ← PWL(pa) – p
        ! Update NRMPL if last extant writer
        if PWL(pa) = ϕ do
        {
                NRMPL ← NRMPL + va
                ! Record causally dependent processes
                VT_NRMPL( va ) ← VT_p
        }
    }
}
```

Figure 8: Checkpoint code

4.5 Process checkpoint of process p:

The algorithms shown above track causality by tracking page accesses. The PRL and PWL capture this information for resident pages with the NRMPL and PALs tracking causality for non-resident pages. As memory sizes continue to grow, the NRMPL and PALs should not be heavily utilised. When a process makes a checkpoint request, five actions need to be performed. Firstly, the vector time of the process must be brought up to date. This is performed using the *COMMUNICATE* procedure shown in Figure 6. Next those pages that are part of the checkpoint which are not swapped out need to be made stable. Once this has been achieved the checkpoint is labelled with the vector time of the process. The fourth step is to reset the PAL since the swapped out pages are now part of a checkpoint. The fifth and final step is to invalidate the checkpointing process' copy of the pages that have been checkpointed. This ensures that the process will fault on these pages when (if) they are next accessed and that proper causal tracking is performed. This does not require the pages to be removed from main memory. This algorithm has been expressed in the manner shown for clarity and many of the actions may be performed in parallel.

4.6 Distributing processes and pages

The operations described above support optimistic process checkpointing on a single node. Although not described here, the mechanism scales to a distributed context. Two additional operations are required: support for reading pages from remote nodes and the migration of processes to other nodes. These operations fit easily into the framework described above. Fetching data from and sending data to remote nodes is very like placing and retrieving pages from swap space. Remote data needs to be tagged with a vector time when it moves between machines. Indeed, most causality tracking schemes model message receipt and transmission rather than take a process based approach. When a process migrates it needs to carry an up-to-date vector time to the remote node. Therefore before migration, its vector time needs to be updated in the manner described in Section 4.5 above. The final required extension is that the pages modified by a process on all nodes need to be checkpointed when a process checkpoint occurs.

5. An example

To illustrate the techniques described above consider a very simple example: that of readers and writers. In this example we assume that two concurrent processes P1 and P2 are accessing a single virtual page V1. The writer P1 increments a value on page V1 which requires an initial read. Process P2 merely reads the value. To simplify the example it is assumed that each process checkpoints after each increment or read operation. In practice checkpoints would not occur as frequently. One possible execution trace of this system is shown in Figure 9.

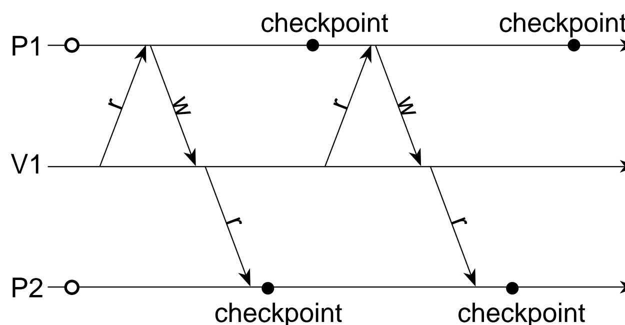

Figure 9: Readers and writers

We assume that the processes start in a consistent state shown by the open circles on the time line in Figure 9.

The trace shown in Figure 9 will result in P1 faulting twice (one read and one write), and P2 once against page V1. The application of the fault handing algorithms described above results in the PRL for the physical page corresponding to V1 containing P1 and P2 while the PWL contains P1. At this point P2 initiates a checkpoint operation which results in the lazy update of the vector time of P2. The state of P2 including the page V1 is checkpointed and labelled with the vector time of P2 <3,3> as shown in Figure 10. At this point in time the latest checkpoint for P1 is <1,⊥> and a new globally consistent state cannot be formed. The last step in checkpointing is to prevent further access to the checkpointed page(s); thus page V1 is invalidated with respect to P2.

Next, process P1 initiates a checkpoint operation resulting in the state of P1 including the page V1 being checkpointed. This state is labelled with the vector time <4,⊥>, creating a new globally consistent cut involving the latest checkpointed states of both process P1 and P2.

Processes P1 and P2 continue to execute repeating the same sequence of events. Notice that the stable state of process P2 cannot become part of a globally consistent state until P1 performs a checkpoint.

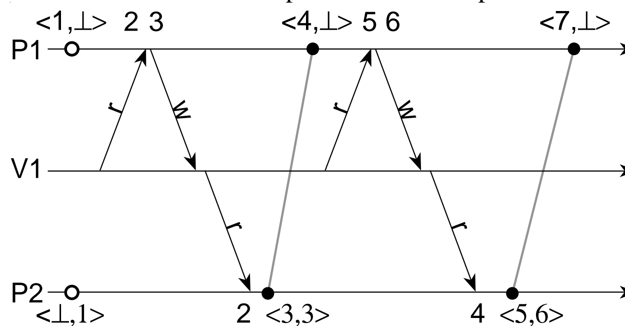

Figure 10: Readers and Writers with Vector Times

6. Implementation Considerations

We are currently implementing the mechanisms described in this paper in the context of the Grasshopper operating system [3]. Indeed, the

motivation for this work was to improve checkpointing in Grasshopper. In Grasshopper, all processes (*loci* in Grasshopper parlance) execute in the context of persistent data repositories called *containers*, and may access any data from any container for which they hold a capability. Consequently, there is no 1-1 correspondence between processes and address spaces as there is in Unix. This computational model is similar to that found in SASOS [2], and Grasshopper may be considered to be a superset of the SASOS approach [12]. In both SASOS systems and systems like Grasshopper, mechanisms such as those described in this paper must be used in order to track page accesses if checkpoints are to take place on any granularity finer than all of RAM. The vanilla Grasshopper memory management system provides no such support [11], and therefore gives a good base-line for the costs involved in this form of causality tracking.

In Grasshopper, each process (locus) is represented by a kernel level data structure, the address of which is used internally by the kernel to uniquely identify the process. The vector time for the process is referenced by this data structure. Although vector times conceptually have as many entries as there are processes in the system, in practice they may be efficiently represented as lists of pairs consisting of the identity of a process and a scalar time.

Each frame of physical memory is represented in the kernel using a C structure called a `Page`. In our implementation, there is no single PRL or PWL data structure. Instead, each `Page` contains two fields, `PRL` and `PWL`, each of which are arrays of process identifiers. Some of the operations described above require all the physical pages accessed by a process to be found. For this reason, an additional per process list is maintained, which contains pointers to the `Pages` accessed by the process. A final complication in Grasshopper is that a process may access pages from multiple containers. As this is not the subject of this paper, we shall not dwell on it here.

In the vanilla (unoptimised) Grasshopper system running on a DEC Alpha 3000/400 (125MHz), it takes 31µs to service a page fault. The mechanisms described in this paper add an additional 8µs to this time. Of this time, 3µs is spent updating the PR/WL, 3µs is spent adding the `Page` to the process list and a final 2µs is spent checking if the page is in the NRMPL. In most operating systems, some of this *extra* work would need to be performed in any case to track accesses to memory. For example, in order to support synchronous checkpointing, the update to the PR/WL (3µs) would be required.

We hope to have some concrete measurements on the time taken to perform vector time updates on checkpoint and invalidation in the near future. We do not expect these to be onerous.

7. Conclusions

This paper presents a new approach to process checkpointing designed to support persistent object systems. It may be seen as an extension to the synchronous checkpointing technique described by Jalili and Henskens and draws on the work of Johnson and Zwaenepoel. The technique is designed for use in an operating system environment such as SASOS or Grasshopper, where processes and address spaces are not synonymous. In such environments some mechanisms must be provided to track access to data by processes.

One important consideration is the granularity at which tracking is performed: it could be at a segment level in a SASOS system or at an object level in object oriented systems. However, in systems like Grasshopper that are intended to support large objects containing a number of small fine grain objects, causal tracking at a page level is an attractive option.

The technique maintains vector times associated with processes. For efficiency, vector times are updated lazily – normally at the time of checkpoint. Preliminary measurements show that the data structures necessary to allow vector times to be updated lazily may be maintained at low cost. The system is applicable to distributed systems in which processes and data may move between machines.

Finally, the system may be augmented with policies to tune performance. For example, the techniques described by Wu and Fuchs may be used in conjunction with our technique to limit roll back. Similarly, policies may be introduced to perform synchronous checkpointing if a consistent cut has not been established in some acceptable time frame; this again limits roll back.

We are currently implementing the techniques described above in the context of the Grasshopper operating system and hope to be in a position to fully demonstrate the efficacy of the technique in the near future.

Acknowledgements

This paper benefits from discussions with John Rosenberg and Anders Lindström of Sydney University. Francis Vaughan made several helpful comments from a great distance during the development of the algorithms. The implementation benefits from the many hours of work Anders Lindström has put into the Grasshopper kernel. Finally we would like to thank Alex Farkas for his many comments on the paper.

References

1. Challis, M. F. “Database Consistency and Integrity in a Multi-User Environment”, *Databases: Improving Usability and Responsiveness*, Academic Press, pp. 245-270, 1978.
2. Chase, J. and Levy, H. “Sharing and Protection in a Single Address Space Operating System”, *Transactions on Computer Systems*, vol 12, 2, pp. to appear, 1994.
3. Dearle, A., Bona, R. d., Farrow, J., Henskens, F., Lindström, A., Rosenberg, J. and Vaughan, F. “Grasshopper: An Orthogonally Persistent Operating System”, *Computer Systsms*, vol Summer, 1994.

4. Elnozahy, E., Johnson, D. and Zwaenepoel, W. “The Performance of Consistent Checkpointing”, *11th Symposium on Reliable Distributed Systems*, Houston, Texas, IEEE, pp. 39-47, 1992.
5. Fidge, C. “Timestamps in Message-Passing Systems That Preserve Partial Ordering”, *11th Australian Computer Science Conference*, University of Queensland, pp. 56-66, 1988.
6. Henskens, F. A., Rosenberg, J. and Keedy, J. L. “A Capability-based Distributed Shared Memory”, *Proceedings of the 14th Australian Computer Science Conference*, Sydney, Australia, pp. 29.1-29.12, 1991.
7. Jalili, R. and Henskens, F. A. “Using Directed Graphs to Describe Entity Dependency in Stable Distributed Persistent Stores”, *Hawaii International Conference on System Sciences*, 1994.
8. Johnson, D. “Efficient Transparent Optimistic Recovery for Distributed Application Systems”, Computer Science, Carnegie Mellon University, Technical Report CMU-CS-93-127, 1993.
9. Johnson, D. and Zwaenepoel, W. “Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing”, *Journal of Algorithms*, vol 11, 3, pp. 462-491, 1990.
10. Lamport, L. “Time, Clocks, and the Ordering of Events in a Distributed System”, *CACM*, vol 21, 7, pp. 558-565, 1978.
11. Lindstrom, A., Dearle, A., Bona, R. d., Farrow, J., Henskens, F., Rosenberg, J. and Vaughan, F. “A Model For User-Level Memory Management in a Distributed, Persistent Environment”, *17th Australian Computer Science Conference in*, *Australian Computer Science Communications*, vol 16, pp. 343-354, 1994.
12. Lindstrom, A., Rosenberg, J. and Dearle, A. “The Grand Unified Theory of Address Spaces”, *Hot Topics in Operating Systems (HotOS-V)*, Seattle, 1995.
13. Lorie, R. A. “Physical Integrity in a Large Segmented Database”, *Association for Computing Machinery Transactions on Database Systems*, vol 2, 1, pp. 91-104, 1977.
14. Mattern, F. “Efficient Distributed Snapshots and Global Virtual Time Algorithms for Non-FIFO Systems”, 1990.
15. Rosenberg, J. and Abramson, D. A. “MONADS-PC: A Capability Based Workstation to Support Software Engineering”, *18th Hawaii International Conference on System Sciences*, pp. 515-522, 1985.
16. Schwarz, R. and Mattern, F. “Detecting Causal Relationships in Distributed Computations: In Search of The Holy Grail”, Department of Computer Science, University of Kaiserslautern and University of Saarland, Technical Report SFB 124-15/92, 1992.
17. Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. “Casper: A Cached Architecture Supporting Persistence”, *Computing Systems*, vol 5, 3, California, 1992.
18. Wu, K.-L. and Fuchs, W. K. “Recoverable Distributed Shared Virtual Memory”, *IEEE Transactions on Computers*, vol 39, 4, pp. 460-469, 1990.