Wanted: An Application Aware Checkpointing Service

Colin Allison

Division of Computer Science,
University of St Andrews, St Andrews KY16 9SS
SCOTLAND

ca@dcs.st-andrews.ac.uk

Checkpointing is an essential component of many application frameworks but OS kernels rarely offer assistance with this task although they are in a very good position to do so. OS process management could and should be extended to accommodate this common need and provide an application aware system service.

1. Introduction

For the purposes of this paper we distinguish between three distinct entities: application frameworks; user programs that run within these frameworks; and OS services. *Application frameworks* include programming languages, databases, and diverse custom systems such as Arjuna[1] and IGOR[2]. They are of particular relevance to the SIGOPS'94 theme[1] because in adapting to existing OS services they highlight the shortcomings of current OS provision. This paper argues that application level *local checkpointing* is such a common need that it should routinely be catered for by an efficient OS level service. Algorithms and issues associated with *distributed checkpointing* [3,4,5] are only of limited concern here in as far as local checkpointing is an essential part of distributed checkpointing and also of certain styles of distributed computation. *Virtual time* for example relies heavily on local checkpoints for distributed synchronisation.

Many application frameworks require a checkpointing service[1,6,7,8,9,10,11,12,13,14,15,18]. Its implementation however, without direct OS support, can incur substantial costs in terms of development time and runtime overhead. On the other hand, OS kernels, be they micro or monolithic, implement process management functions such as scheduling and context switching as efficiently as possible. These functions, naturally, involve the saving and restoring of state. That is, they effectively checkpoint a process at every context switch. If the OS offered access to its privileged state management capabilities in the form of a checkpointing service then developers can be helped with a time consuming task, and runtime overhead can be reduced. For example, an average Condor checkpoint which has no OS support takes over a minute[9], while an average Unicos checkpoint using its `checkpoint(2)` system call takes about ten milliseconds[14]. In both cases the outcome is an executable file. However, as Condor runs on standard Unix workstations and Unicos runs on Cray supercomputers, a fairer measure of the OS/application mismatch might be the number of OS context switches an application undergoes while carrying out a checkpoint, normalised against system load.

The reminder of this paper is organised as follows: Sections 2 - 4 support the case that checkpointing requires direct OS support, Section 5 outlines an OS checkpointing service, and Section 6 concludes. Related work is referred to throughout the paper.

---

[1] "Matching Operating Systems to Application Needs"

2. Who Uses Checkpointing and Why

This section briefly lists some of the many and diverse frameworks which use checkpointing, and their motivations for doing so. It shows that there is a common need for a checkpointing service.

Application frameworks which rely on checkpointing include:

- transaction based programming[1,6,13], and databases in general[12],
- persistent object systems[7,10,16],
- virtual time based distributed computation[6, 15],
- job control systems for long running programs[9,14],
- replay debugging[2,17],
- distributed simulation[18,19].

Motivations for checkpointing include:

- concurrency control and atomicity[1,6,12],
- rollback and backtracking[2,6,12,15,18],
- failure recovery and fault tolerance[6,11,12,20],
- process and object migration[21,22],
- reversible execution[2,17].

3. State Management in a Virtual Time Framework

Virtual time frameworks are of particular interest because they make the heaviest demands on state management. We distinguish between *implicit virtual time* (IVT), for example, WARP[6], and *explicit virtual time* (EVT), for example, TWOS[19]. In both, objects rely on the ability to backtrack when progress has been over optimistic. The demands on state management functions in EVT frameworks can be so great that special purpose operating systems have been developed[19] and custom hardware designed[23] in order to satisfy those demands. Although IVT is more economical in its state saving, hardware assistance for its specialised memory management needs is still attractive[24]. In the WARP framework[6] IVT is the basis of a uniform coherence mechanism in distributed systems. When operating in optimistic mode (the default) a WARP object may run ahead of its collaborators. If its optimism proves ill-founded then the branch of a distributed computation that is visiting that object must backtrack to an earlier state and proceed along a different path. The side effects of an object's optimism are limited to state changes, which are resolved through restoration from a checkpoint, and messages sent to other objects, which are cancelled by sending *antimessages*. There may be several active object clones concurrently representing different aspects of possible futures for an object and thus several alternative states must be maintained. As all WARP protocol decisions are made *locally*, the efficiency of local checkpointing is a major factor in the performance of the framework. Although it is possible to run WARP within a common OS environment[2] through the judicious use of memory mapped files, better OS support for checkpointing would greatly simplify the framework's state management routines and reduce runtime overheads. It follows that such OS support would also be of benefit to many application frameworks.

---

[2] SunOS 4.1.3

4. Checkpointing Issues and Costs

There are three main issues:

- How is *state* defined, represented and encapsulated?
- Where does responsibility lie for state definition and checkpoint initiation?
- What costs are involved?

4.1 State Definition

State must be defined before it can be encapsulated, and must be encapsulated before it can be represented as a checkpoint. Applications and operating systems discern different views of an application's state. In particular the OS view of a *distributed* application's state can only be compared with that applications own *local* view. States associated with an OS process, or process group, are essentially a superset of an application's local state and include many details which are not meaningful to the application user. For example a conventional OS process control structure[25] contains information about resource accounting and priority level, registers, page tables and swap areas. A key issue is how to map between the application and the OS views of state and is routinely achieved through a symbol table.

A *full* checkpoint is a record of an entire application in a self-consistent state. If the application resides entirely in a *local* environment, that is, a closely-coupled architecture, then efficiently computing the checkpoint is feasible[5]. For example, a restartable Unicos checkpoint can be taken of an entire job, which may include multiple processes, open files and open sockets. Checkpointing a distributed application is more problematical[4] and the question arises as to what a local checkpoint should contain besides state variables, I/O channels and the program counter value. If all non-local I/O is restricted to messages then all channel histories since the last checkpoint, as well as the current channel states, may need to be included. However, encapsulation and message logging do not fully solve the problem as I/O may have caused non-reversible effects, that is, actions that have *committed*. If an application has no clear semantic policy on commitment then it is anarchic and cannot be meaningfully checkpointed.

4.2 Responsibility and Transparency

Certain frameworks such as Arjuna[1] ask the programmer to supply *save* and *restore* routines which serve to both define and encapsulate state. At the other end of the spectrum *transparency* is considered paramount because no extra work is required from the programmer. Condor[9] and IGOR[2], for example, checkpoint user programs automatically. Of course there are many variations along this spectrum. In TWOS[19] the programmer must syntactically isolate state variables in the source code, but the framework then performs checkpoints automatically. Note that *transparency* in checkpointing can lead to noticeable delays. The KeyCOS kernel[8] requires no work from an application framework or a user program. It checkpoints its entire system wide state periodically, but can take twenty seconds or more during which time the entire system is halted and all applications pay the same high price, regardless of need. Checkpointing in frameworks such as Condor [con] suffer from poor performance because the service is transparent to both the user program and the OS. On the other hand, Condor's success is due to its convenience. The Unicos based framework described in [14] offers the best of all worlds. It takes full responsibility for automatic checkpointing and benefits from OS kernel support, but allows the user to override default parameters such as checkpointing frequency. To summarise: if there is no OS support transparent policies fare worse, but with OS support all policies benefit.

4.3 Checkpointing Costs

Costs are characterised by time and space requirements. There are two types of timing information of interest: how long does the checkpoint take in total and how long is the application incapacitated for? Both CPU and elapsed time are of interest. Many schemes attempt to checkpoint asynchronously to minimise interference with the application. The total time is still of importance in such systems as it constrains the frequency of checkpointing. Space is determined by the number of checkpoints that are required to be kept and the state size. If the framework's purpose is simple fault tolerance then this is twice the state size on stable storage. However, in *virtual time* and *replay debugging* several generations are kept and space management becomes a major issue.

Both time and space costs can be cut by using an *incremental* state saving scheme. *Logging* is an example of a coarse-grained incremental scheme widely used in databases to complement full checkpointing[26,12]. While its granularity suits database transactions it has the drawback of being too intrusive in other frameworks. However, if a scheme exploits existing virtual memory mechanisms and tracks state changes at the page level then considerable savings can be made because the granularity of change is aligned with the OS. IGOR[2] does exactly this, only resolving increments into a full checkpoint when required. Feldman and Brown stress that IGOR could not have been implemented without the introduction of a novel system call – `pagemod(2)` – which returns a list of pages modified since it was last called. Another incremental scheme of note is described by Fujimoto in[23]. Part of the virtual address space is reserved solely for state saving purposes – version controlled memory – and is maintained as a stack of incremental state changes. Custom hardware in the form of an application specific integrated circuit (ASIC) called the "rollback chip" is deemed necessary to implement version controlled memory[23]. If hardware must be used then the Field Programmable Gate Array (FPGA) is an attractive alternative to the ASIC as its functionality can be routinely changed. Allison and Livesey propose the use of FPGAs for experimenting with state management policies in[24].

5. Towards an OS Checkpointing Service

The OS optimises its state management by working directly with memory management hardware, an option normally denied to application frameworks. There are however precedents for OS process control blocks containing application specific information e.g. signal handlers and advised virtual memory behaviour[25]. Microkernels such as Mach hold out great promise by inviting users to supply *policy* for functions traditionally hidden in the kernel, for example, memory management. However, there is no great willingness on the part of the kernel to let an application interfere in its traditional domain. Although Mach 3 allows for user supplied memory managers, application directed checkpointing may have to play second fiddle to the kernel. For example, with reference to its page replacement algorithm, "even external memory managers have no influence; if they do not respond fast enough to a request to write a page, the default memory manager will be used"[27]. So much for open systems. Yet there is not necessarily a conflict here – what we want is for the memory and process management algorithms to accommodate an application specified checkpointing policy – exactly how this is done can remain hidden. In micro kernels application supplied policies must be promoted to a much higher status, as in the proposed SPIN system[28].

In monolithic kernels the process control block could be extended to contain some of the following application-supplied features and options:

- an indication that the application is interested in checkpointing
- a list of memory regions to be checkpointed
- a list of destinations for the checkpoint(s)
- a destination in user address space for placing receipts
- the number of distinct checkpoints to preserve
- an advisory frequency for automatic checkpointing
- a request to checkpoint as soon as possible
- a need for assurance that the checkpoint is stable
- a request to restart from a previous state

6. Conclusion

We have argued that operating systems can usefully help application developers by providing a state checkpointing service and have shown that there is a widespread need for such a service. As the OS is already heavily involved in the business of state management it can exploit its memory management capabilities to checkpoint incrementally, thereby keeping costs to a minimum. We have pointed to some relatively rare examples of existing OS checkpointing support[2,14,19] which illustrate its benefits in terms of functionality, convenience and performance for both application frameworks and programmers. None of these examples however constitute a satisfactory general purpose service so we have outlined some of the parameters that a uniform service might include.

7. References

1. CAUGHEY S.J., SHRIVASTAVA S.K., & MCCUE D., Implementing Fault-tolerant Object Systems on Distributed Memory Multiprocessors. **In** *Proc. 2nd IEEE International Workshop on Object Orientation in Operating Systems*, Paris, September 1992, pp. 172–179.
2. FELDMAN, S., & BROWN, C., IGOR: A System for Program Debugging via Reversible Execution. *ACM SIGPLAN* **24**,1 (January 1989) 112-123.
3. JOHNSON, D. & ZWAENEPOEL, W., Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. *Journal of Algorithms*, **11**,3, (1990) 462-491.
4. MATTERN F., Efficient distributed snapshots and global virtual time algorithms for non-FIFO systems (private communication), March 1990. (To appear in the Journal of Parallel and Distributed Computing).
5. PLANK, J. S., Efficient Checkpointing on MIMD Architectures. PhD Thesis, University of Princeton, Deptartment of Computer Science, Technical Report 406 (1993).
6. LIVESEY M.J. & ALLISON C., Coherence in distributed persistent object systems. **In** A.Albano & R.Morrison (eds.) *Persistent Object Systems*, *San Miniato 1992* (Proc. 5th Intern. Workshop on Persistent Object Systems, San Miniato, Italy, September 1992), Springer-Verlag (1993) 186–197.
7. BROWN A.L. & ROSENBERG J., Persistent object stores: an implementation technique **in** *Implementing Persistent Object Bases: Principles and Practice*, Morgan-Kaufmann (1991) 199–212.
8. LANDAU C.R., The Checkpoint Mechanism in KeyKOS, **In** *Proc. 2nd IEEE International Workshop on Object Orientation in Operating Systems*, Paris, September 1992, pp. 86–91.
9. LITZKOW M. & SOLOMON M., Supporting Checkpointing and Process Migration Outside the Unix Kernel. *Proc. Usenix Winter Conference*, San Francisco, 1992.

10. OLSEN, M.H. A Persistent Object Infrastructure for Heterogenous Distributed Systems. *Proc. 2nd IEEE International Workshop on Object Orientation in Operating Systems*, Paris, (Sept. 1992), 49-56.

11. BIRMAN, K., COOPER, R., JOSEPH, T., MARZULLO, K., MAKPANGON, M., KANE, K., SCHMUCK, F., & WOOD, M., The ISIS System Manual, Version 2.1. , *The Isis Project* (1990).

12. BERNSTEIN, P.A., HADZILACOS, V., & GOODMAN, N., Concurrency Control and Recovery in Database Systems. *Addison-Wesley* (1987).

13. LISKOV, B., Distributed Programming in Argus, CACM 32, 3 (March 1988) 300-312.

14. ATTIG N., & SANDER V., Automatic Checkpointing of NQS Batch Jobs on CRAY UNICOS Systems. **In** *Proc. Cray User Group Meeting*, Montreux (Spring 1993).

15. JEFFERSON D.R., Virtual time, *ACM TOPLAS* **73** (July 1985) 404–425.

16. DEARLE A., DI BONA R., FARROW J., HENSKENS F., LINDSTROM A., ROSENBERG J. & VAUGHN F., Grasshopper - A Persistent Operating System for Conventional Hardware. *Proc. 2nd IEEE International Workshop on Object Orientation in Operating Systems*, Paris, (Sept. 1992), 81–85.

17. PAN, D.Z., & LINTON, M.A., Supporting Reverse Execution of Parallel Programs, *ACM SIGPLAN* **24**,1 (January 1989) 124-129.

18. FUJIMOTO, R., Parallel Discrete Event Simulation. CACM **33**,10 (1990) 30-53.

19. JEFFERSON, D.R., et.al., Distributed Simulation and the Time Warp Operating System. *Proc. of 11th ACM Symposium on OS Principles*, Austin, Texas, SIGOPS Review **215** (1987) 77-93.

20. VAUGHN F., DEARLE A., CAO, J., DI BONA R., FARROW, J., HENSKENS F., LINDSTROM, A., & ROSENBERG J. Causality Considerations in Distributed, Persistent Operating Systems, *Computer ScienceTechnical Report*, Uninversity of Adelaide (1993).

21. BURGESS P., LIVESEY M.J. & ALLISON C., Process migration as an aid to tuning embedded systems, *Proc. Parallel Computing (ParCo) '93*, Grenoble, France (September 1993).

22. NUTTALL, M., Survey of Systems Providing Process or Object Migration. *Department of Computing Research Report DoC 94/10*, Imperial College (May 1994).

23. FUJIMOTO R.M., Design and evaluation of the rollback chip: special purpose hardware for Time Warp, *IEEE Trans. on Comput*. **C-41**,1 (January 1992) 68–81.

24. ALLISON C., & LIVESEY M.J., A Dynamically Configurable Co-processor for Microkernels, *Research Report CS/94/1*, University of St. Andrews, 1994.

25. LEFFLER S.J., MCKUSICK M.K., KARELS M.J. AND QUARTERMAN J.S., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, (1988) 69 - 165.

26. BIRRELL, A., JONES, M., & WOBBER, E. A Simple and Efficient Implementation of a Small Database, *Proc. of 11th ACM Symposium on OS Principles*, Austin, Texas, SIGOPS Review **215** (1987) 149-154.

27. LOEPERE K., *Mach 3 Kernel Principles*, OSF documentation (1992) 55.

28. BERSHAD, B.N., CHAMBERS, C., EGGERS, S., MAEDA, C., MCNAMEE, D., PARDYAK, P., SAVAGE, S., SIRER, E.G., *SPIN* – An Extensible Microkernel for Application_specific Operating System Services. *Dept. of Computer Science Technical Report 94-03-03,* University of Washington, (February 1994).