Comments on "Transparent User-Level Process Checkpoint and Restore for Migration" by Bozyigit and Wasiq

Felix Rauch, Thomas M. Stricker
Laboratory for Computer Systems
ETH - Swiss Institute of Technology
CH-8092 Zürich, Switzerland
{rauch,tomstr}@inf.ethz.ch

Abstract

The simple checkpointing and migration system for UNIX processes as described in the article of Bozyigit and Wasiq [1] can be improved in two ways: First by a technique to checkpoint and migrate applications without the need to recompile them and second by an alternative approach to precisely locate all the data segments of a process that need to be checkpointed. We fully acknowledge the difficulty to do checkpointing or even portable checkpointing for the general case of processes and do not claim to solve the many remaining problems with the simplistic checkpointing and migration approaches presented in the earlier article. Still we are aware of many systems and applications where a simple solution is extremely helpful once it also works with binaries.

1 Introduction

Recent multi purpose and multi boot clusters are not necessarily built out of rack mounted machines, but consist also of users or even students workstations [4]. On such workstations, memory intensive and long running processes cause mainly two kinds of problems: (1) Users might want all the available resources of their workstations and therefore get rid of the resource intensive background jobs and (2) the machines might be rebooted to change the operating system or to perform maintenance tasks. In both cases it is desired that the long running process' state is not lost and therefore the process is *checkpointed*. To proceed the process' work, its state is *migrated* to another available and probably idle machine in the cluster and the process is continued.

There are different approaches to apply checkpointing and migration to self written programs or open source software as the structure of the available source code can be changed to match the chosen checkpointing technique[1]. For other programs there is no source code available, such as e.g. for many commercial applications. These applications can not be checkpointed and migrated with the technique described in the paper of Bozyigit and Wasiq [1], as recompilation is required.

The complete state of a process and the state in the operating system linked to that process remains difficult to understand and hard to capture to its full extent. The precise interaction between a user level process, the libraries it uses and the operating system can be extremely complex and it might be nearly impossible to get the complete picture of all operating system state just from intercepted calls. This leaves the approach as presented by Bozyigit and Wasiq a bit vague and it remains to be seen if it is sufficient to checkpoint and migrate applications that go beyond simple example codes. Nevertheless, the authors have applied their ideas to a similar checkpointing system used primarily in a circuit/device simulation environment in practice [3].

2 Problem statements

The approach taken by Bozyigit and Wasiq requires the inclusion of a header file in all source code files and a recompilation of the applications to be checkpointed. The requirement to have the source code available limits the usability of the approach to applications developed inhouse and open source applications. Existing commercial programs not available under an open source licence can not be checkpointed or migrated.

Their approach also relies on specific compiler techniques: To know the start and the end of the data segment, small 'C' files with special variables are linked before and after the other source files. This approach works only with compilers that order variables in memory in the order they were linked. Future compilers might pool variables together to optimize cache line usage. This optimization could lead to parts of the memory which are not

[1] Some techniques allow cross-platform migration of running processes by transforming and recompiling the source code of an application, such as described in [2].

checkpointed. On some UNIX systems, the data segment does not consist of contiguous, accessible pages. Writable pages containing data segments might be alternated with read only pages of constant data. All these problems lead to a more advanced approach of checkpointing the data parts of a process.

3 Suggested Improvements

To overcome the need for recompilation of applications to enable checkpointing, we suggest to use dynamic linking. Dynamic linking techniques allow to track open files, duplicated file handles and other program state by redirecting system calls to special library routines. Instead of redirecting system calls by introducing macros in the source code, we dynamically link the existing executable to our checkpointing library by the use of the `LD_PRELOAD` environment variable. The system then links our own library before all system libraries. System calls like `open()`, which are usually implemented in the C-library, are automatically redirected to our own library instead. Our library then copies the required information from the parameters of the system call and dynamically links the real C-library by use of the `dlopen()` and `dlsym()` dynamic linker functions available on Solaris. With this improved technique, system calls can be intercepted without changing the original source code.

A more difficult problem is the interception of the `main()` function of the application to be checkpointed. As the `main()` function is in the application binary, it can not be overwritten with a dynamic library. Instead of the dynamic linker we use the audit functions of Solaris. The audit functions allow us to compile another library which is also linked dynamically with application binary before program start. After linking but before the transfer of control to the `main()` function the function `la_preinit()` in our library is called. This function then initializes the checkpointing library. With this approach the checkpointing library can be initialized without changing the source code of the application.

Another suggested improvement is a better method to find the data segments of the process to checkpoint. Instead of trying to link special variables at the beginning and the end of the data area and then checkpointing the memory between the addresses of the two variables, we suggest to analyze the ELF binary of the process. The binary in ELF format contains informations about its different segments. The segments are marked as read only data (like program text and probably constant data) or read/write data (like initialized but not constant data). Read only segments need not be checkpointed as they do not change during the process' lifetime and are automatically loaded into memory when the applications is restarted after its migration. On the other hand the process may change all data in read/write segments, therefore it is best to checkpoint all its read/write segments. This method assures that all non-constant data segments are checkpointed and that there will be no error after migration due to erroneously restored read only segments in the data space of the process.

4 Conclusions

The improvements presented in this paper allow to checkpoint and migrate applications without the need to change the source or even recompile them. This enhancement is especially important for programs for which there is no source code available and allows to use the checkpointing and migration mechanism even with some commercial applications.

In our own experience we found that it is difficult to implement a truly portable user-level checkpointing and migration mechanism. Such a mechanism relies extensively on subtle differences of the various existing UNIX systems and their associated libraries. Remaining problems for user-level checkpointing consist of non-checkpointable state in network sockets and randomly accessed files.

Checkpointing and migration mechanisms help to better utilize the available resources on modern clusters consisting of rack mounted machines or even users workstations. Equiping a cluster with user-level checkpointing facilities is easy and allows owners of large jobs to keep their long running jobs on unused and otherwise idle workstations. Our improvements allow to integrate some commercial applications into the checkpointing and migration facilities without the need to recompile them.

References

[1] M. Bozyigit and M. Wasiq. User-Level Process Checkpoint and Restore for Migration. *Operating Systems Review*, 35(2):86–95, 2001.

[2] Balkrishna Ramkumar and Volker Strumpen. Portable Checkpointing for Heterogeneous Architectures. In *In 27th International Symposium on Fault-Tolerant Computing — Digest of Papers*, pages 58–67, April 1997.

[3] F. Rauch. Porting ckpt_lib to different UNIX operating systems. Internal report, ISE Integrated Systems Engineering, Zürich, Switzerland, October 1996.

[4] Felix Rauch, Christian Kurmann, Blanca Maria Müller-Lagunez, and Thomas M. Stricker. Patagonia — A Dual Use Cluster of PCs for Computation and Education. In *2. Workshop Cluster Computing, Karlsruhe*, pages 65–75, March 1999.