A POLL-FREE, LOW-LATENCY APPROACH TO PROCESS STATE CAPTURE / RECOVERY IN HETEROGENEOUS COMPUTING SYSTEMS

Prashanth P. Bungale and Swaroop Sridhar
Department of Computer Science & Engg.
The National Institute of Engineering
Mysore, India
E-mail: { pbungale, swaroop } @ ieee.org

Vinay Krishnamurthy
Department of Computer Science & Engg.
Vidyavardhaka College of Engineering
Mysore, India
E-mail: vinay_krishnamurthy@rediffmail.com

Abstract

A major issue of process state capture in heterogeneous computing systems is that it cannot simply be initiated instantaneously, once a request for capture has been received. This is because the capture can be initiated only at certain points – at points which have equivalent points in the other instances of the computation on different architectures – so that the process can be restarted at exactly the same point at which it was paused. For ensuring minimum latency, the state capture should be initiated at the very next point of equivalence encountered, once requested. At the same time, it should be ensured that the performance overhead incurred during normal execution should be kept at acceptable levels. This paper proposes a fundamentally new approach to process state capture and recovery which achieves the above objectives.

In a polling approach, to achieve minimal latency (wait-time between capture request and actual initiation), poll-points would have to be placed at all potential points of equivalence. However, the performance overhead incurred due to polling during normal execution would reach severely unacceptable levels in this case. Our solution to the heterogeneous process state capture problem is fundamentally different in that it effectively enables all potential points of equivalence present in a computation, so that minimal latency is ensured.

Key Words

Heterogeneous computing, Process state capture and recovery, Low-latency capture initiation

1. Introduction

With the cost benefits of the mass production of smaller machines, the majority of today's computing power exists in the form of PCs or workstations [1]. Distributed computing systems, comprising of several loosely-coupled computers communicating over a high-bandwidth network, are becoming more and more available to the user community. Some level of heterogeneity is really the norm in such systems. The presence of heterogeneity in such computing systems significantly complicates the design of a process state capture/recovery mechanism, which must automatically capture the state of a running program in some stable form and then restart the program from the point of capture at some later time, possibly on a different platform.

A substantial body of research demonstrates the utility and desirability of such a mechanism. For example, process migration policies supporting load sharing and/or fault tolerance can be based on a process state capture facility (e.g. Condor [2]). Also, one common method of providing reliability for applications is to provide checkpoint and restart functionality. If application state is stored periodically (checkpointed), when some component of the system is lost (due either to failure or to load-balancing requirements) an application can use recently saved computational state to restart without having to regress too far in the computation. Providing fault-tolerance for applications in these systems will be greatly simplified by checkpoint/restart functionality.

Beyond these existing uses for process state capture and recovery, the availability of such a mechanism also opens up new possibilities such as improved resource management, platform-independent debugging using checkpoints and message logs to replay a process from a given point in execution, or statically examining the state of a process as captured in a checkpoint [3]. The increasing importance of the use of process state capture and recovery has made the design of a such a mechanism a key research issue in heterogeneous computing.

2. The Heterogeneous Process State Capture Problem

A mechanism solving the heterogeneous process state capture and recovery problem must provide the ability to generate a checkpoint for an *active process* – a complete description of that process's state and point in execution – and also support the later use of that checkpoint to restart a process with equivalent state and at an equivalent point in execution, possibly on a different platform from the one on which the original checkpoint was created [4].

A major issue of process state capture in heterogeneous computing systems is its initiation, once the request has been received. The process cannot simply be paused (for capturing its state) at any point of its execution, but can only be paused at points, which have *equivalent points* in all the other instances of the same computation on different platforms [5]. For achieving minimum latency, the state capture should be initiated at the very next point of equivalence encountered.

Also, the possibility of cross-platform recovery leads to the most fundamental solution constraint: the mechanism should capture the state in a *platform-independent* manner – i.e., checkpoints produced on a computer system of any architecture should be recoverable on a system of any other architecture [6]. For example, one straightforward approach is to use an interpreted language. In these designs, the interpreter acts as a virtual machine that can artificially homogenize a system composed of heterogeneous elements. Unfortunately, such schemes severely compromise performance since they run at least an order of magnitude slower than their native code counterparts. Therefore, in our intended environment, processes run on nodes, typically executing in native code form due to performance considerations.

In homogeneous systems, process state capture and recovery mechanisms can simply and directly copy the state of a process *verbatim*, without semantic analysis and interpretation of that state. Unfortunately, in a heterogeneous environment, the state of a process cannot be captured using this naï ve approach because of differences in instruction sets and data representation. To mask the varying features of a process's environment in a heterogeneous system, a state capture mechanism must examine and capture the *logical* internal structure and meaning of the process state – i.e., the logical point in execution, the call stack (or call stacks, if threads are supported), complex data structures, the logical structure and contents of heap allocated memory, and all other process state must be analyzed and captured in a *platform-independent* format.

3. Related Work

Although process state capture/recovery mechanisms for homogeneous computing systems are well-developed and can now typically be performed with minimal latency and overhead, much less progress has been made towards providing such a functionality across heterogeneous architectures. Because of the additional inherent complexity introduced by heterogeneity, very few designs for such a facility have been developed to date.

The Tui system [7] has been constructed to provide a heterogeneous migration tool for use on four common architectures within the Unix environment. In this system, the capture (and recovery) of the activation history state of a process is performed in a highly machine dependent manner, with full knowledge of the particular idiosyncratic conventions followed on each of the target platforms.

The process introspection model proposed by Ferrari [4] and the portable checkpointing model presented in [8, 9] overcome the above drawback by performing the capture of the activation history state in an entirely machine-independent manner, using the call-return semantics provided by the high-level programming language or by the intermediate instruction set. Here, the architectural differences will be taken care of by the compiler. However, almost all existing systems, including these systems follow a polling approach to initiate the process state capture. *Poll points* are introduced into the execution where the process determines if a capture should be initiated.

There are many *possible* points of execution equivalence that can be identified in any program. But, in a polling approach, not all of these candidates would be effectively enforced as *actual* points of execution equivalence – i.e., only the points at which poll-points are introduced would be enabled as actual points of equivalence. Thus, poll-points can be considered as a pre-defined subset of the entire set of points of equivalence present in a computation. The selection of this subset is based on a trade-off between the performance overhead incurred during normal execution and the wait-time from request to actual initiation of the capture.

In such a polling approach, once a request for capture has been received, it is initiated only at the next *poll-point* encountered. Therefore, the capture would be actually initiated after an arbitrarily long period of time. In applications such as process migration due to load-balancing policies, or logging mechanisms for fault tolerant systems, such latencies would not be acceptable. In the case of load balancing, for example, the very purpose of migrating the process itself is to reduce the load on the system as soon as possible.

4. Our Approach

We now describe our poll-free, low-latency approach to constructing a heterogeneous process state capture/ recovery mechanism. This is a fundamentally different approach than existing solutions to the process state capture problem, which typically are based on a polling approach.

4.1. Objectives

a. Minimal Latency: The state capture mechanism should ensure minimum possible latency (the time delay between when a capture is scheduled or requested and when the capture is actually initiated) which is the time taken to reach the very next point of equivalence in the computation.

b. Ease of Use: When possible, the mechanism should be fully automatic, requiring little or no effort on the part of the application programmer. Such full automation should be possible for programs expressed in a platform-independent manner, i.e. programs that do not rely on specific hardware or software features of certain computer systems.

c. Generality: The mechanism should be appropriate for use with a wide range of architectures and a wide variety of programs that are written in a variety of languages, and that solve a wide range of problems.

d. Efficiency: As a goal, in addition to providing efficient state capture and recovery, the mechanism should introduce acceptable run-time performance overhead. In particular, if checkpoints are not performed during a certain period of execution, a process with state capture and recovery service available to it should not run significantly slower than the version of the code without this service available over the same period.

4.2. Solution Overview

Our solution to the heterogeneous process state capture problem is fundamentally different in that it effectively enables all potential points of equivalence present in a computation, so that minimal latency is ensured. In the polling approach, to achieve this minimal latency, poll-points would have to be placed at all potential points of equivalence. However, the performance overhead incurred due to polling during normal execution would reach severely unacceptable levels in this case. Therefore, we propose a poll-free solution.

One of the major aspects of our design is the modification of a program to incorporate state capture and recovery functionality, giving processes the ability to autonomously save and restore their states, once initiated. We assume that the process is based on a program that is either written in or has been translated to an imperative, stack-based intermediate representation to which our transformations will be applied – likely by a compiler, but also possibly by a programmer.

The key element of our design is a table which maps all the points of equivalence to the corresponding points in object code for each target architecture, obtained using compiler-support. This table is primarily used to capture – in a machine-independent manner – the current execution point of the process and all the points where execution was frozen due to function activations. These points are then mapped onto their corresponding points in the destination architecture's object code during recovery. Also, once a request for state capture has been received, this table is used to place control-flow instruction(s) such that it would be ensured that the state capture is initiated at the very next point of equivalence encountered.

Certain parts of the process state are easily captured – for example, any global variable or heap allocated data structures, being globally addressable, are easy to capture and recover. The key difficulty in capturing the process state is the capture of the activation history state. In our approach, the process utilizes the native "function return" mechanism provided by the intermediate instruction set to capture the stack state. The currently active function saves its own state (which only it can access) and returns to its caller, which in turn saves its own state, and so on, until the stack capture is complete. Similarly, to effect recovery, the process employs the native "function call" mechanism provided by the intermediate instruction set. During recovery, the base function restores its state, and then calls the next function in the captured stack, which repeats this process until the stack is completely restored. To preserve program correctness (so that the function call sequence repeated during recovery does not produce any side-effects), the given program shall be transformed such that all function calls appear only in simple expression statements, which are side effect free.

The activation history capture/recovery mechanism described above is accomplished by adding epilogues in each function (which is non-reachable during normal execution but is reached only during capture/recovery), for both saving and restoring the activation state.

4.3. Assumptions

a. Compiler support is available for obtaining the equivalence point table.
b. Operating system support is available for obtaining the current value of the program counter for a process.
c. Machine dependent optimizations across points of equivalence shall not be allowed since they may prevent the different compiled versions of the program across heterogeneous platforms from hitting every point of equivalence consistently.

4.4. State Capture Algorithm

When a request for capture of a process's state is received from an external agent, the following steps are carried out:

a. The current value of the program counter is obtained using operating system support.
b. This value is used to identify the currently executing function by looking up a function to code-range map-ping table.
c. The part of the equivalence point table that belongs to this function is searched to find the current instruction with respect to the intermediate representation.
d. The flags corresponding to the current statement – flags are set by the compiler for each statement indicating whether it contains a call instruction, a return instruction, or any other control flow instruction and also indicating whether it is a jump destination – are examined.

e. We now place a call instruction to the state capture initiator function at an appropriate point of equivalence so that control can be passed to the state capture mechanism.
   1. If no control flow instructions are indicated to be present within the current statement, we place the call instruction at the point of equivalence lying sequentially next to the current execution point.
   2. Else If the presence of a call instruction is indicated to be pending, we place the call to the initiator at the current execution point itself. In this case, we are initiating state capture at the previous point of equivalence itself. This is because the modifications made to a function call expression guarantee that we can repeat the instructions already executed since the previous point of equivalence, without causing any side-effects, on recovery.
   3. Else If the presence of a return instruction is indicated, the call instruction is placed at an equivalence point lying sequentially next to the return's destination point.
   4. Else If any other control flow instruction is found to be present, the call instruction is placed at all the points indicated as jump destinations within the current function.

   The process is now "informed" to initiate the state capture as soon as possible.
f. When the process starts executing later, the call instruction gets executed eventually, and the control is thus transferred to the initiator function.
g. Once the control is transferred to the state capture initiation function, the following tasks are performed:
   1. The point of equivalence at which (and the interrupted function within which) capture has been initiated is noted.
   2. The return address of the current activation record is replaced by the address of the saving epilogue of the interrupted function. Finally, a return is performed so that control is passed to that saving epilogue.

   The initiation of state capture is now complete.
h. The saving epilogue performs the following tasks:
   1. Save the local variables and actual parameters present in the current activation record.
   2. Identify the caller of the current function, by looking up the function to code-range mapping table using the return address available in the current activation record. The point of equivalence preceding the point of execution of the calling function is noted.
   3. The return address of the current activation record is replaced by the address of the saving epilogue of the caller function so that when this epilogue performs a return, control is passed to the saving epilogue of the caller function, after cleaning up the current activation record.
i. Step h is repeated until all activations are saved.
j. When the bottom of the activation history stack is reached, the epilogue also performs the saving of the static (global) data and the heap data.

While saving the activation, static or heap data, the state of a pointer is captured according to its logical meaning (i.e., the data object or code point to which it is pointing) rather than its value indicating the physical address [10, 11, 12]. This capture requires *run-time support* (for registering local, global and heap data) in order to identify the data object being pointed, given a physical address. For all other data types, the data is copied verbatim into the checkpoint.

4.5 State Recovery Algorithm

Once a new process has been created on the destination machine, the following steps are carried out to perform the process state recovery:

a. A jump instruction with destination as the corresponding *restoring epilogue* is placed at the entry point of each function present in the program.
b. When the first activation record (for the base function) is created, the corresponding epilogue will restore the static (global) and heap data from the checkpoint file.
c. The restoring epilogue performs the following tasks:
   1. Restore the local variables and actual parameters into the current activation record.
   2. If there are no more activations to be restored, the original entry point of each function is restored back.
   3. A jump takes place to the point in the destination's object code corresponding to the point of execution (with respect to the intermediate representation) noted during state capture.
d. Step c is repeated until all activations have been restored. This happens automatically since the point to which the jump takes place will contain a call instruction until all the activation records have been restored.
e. The process finally resumes normal execution.

Again, while restoring the activation, static or heap data, the state of a pointer is mapped back from its logical meaning to its value indicating the physical address on this machine. For all other data types, the source data copied into the checkpoint is now translated accordingly into the destination architecture data format by using data format mapping functions, if necessary. This is in accordance with the "receiver makes right" policy. This policy has the advantage that only one translation needs to be done (by the receiver) and there is no need for any intermediate data format representation. Also, if the state is to be recovered on the same architecture as the source, then there is no need of any translation at all.

5. Conclusion

The approach to process state capture and recovery in heterogeneous computing systems presented in this paper ensures that the state capture is initiated at the very next point of equivalence encountered, once the request for capture has been received. This results in the *least possible latency*. At the same time, by making the approach *poll-free*, we ensure that there is absolutely no performance overhead incurred due to polling during normal execution. The overhead incurred is limited to the run-time support required for the registration of *dynamic data* (including activation and heap data) and the performance enhancements lost due to hindering machine-dependent optimizations.

6. Acknowledgement

The authors are deeply indebted to Dr. Sreenivas T H (Asst. Professor at the Department of Computer Science and Engineering at the National Institute of Engineering) for his suggestions, guidance and help rendered during the course of our work.

References

[1] Ramon Lawrence, "*Survey of Process Migration Mechanisms*" Student Report for Course Project, 1997, University of Manitoba.

[2] M.J. Litzkow, M. Livny, and M.W. Mutka, Condor—A Hunter of Idle Workstations, *Proceedings of the Eighth International Conference on Distributed Computing Systems*, 1988, pp. 104-111.

[3] Adam John Ferrari, "*Process State Capture and Recovery in High-Performance Heterogeneous Distributed Computing Systems*," Ph.D. Thesis, Department of Computer Science, University of Virginia, January 1998.

[4] Adam J. Ferrari, Stephen J. Chapin, and Andrew S. Grimshaw, "*Process Introspection: A Heterogeneous Checkpoint / Restart Mechanism Based on Automatic Code Modification*," Technical Report CS-97-05, Department of Computer Science, University of Virginia, 1997.

[5] David G. Von Bank, Charles M. Shub, and Robert W. Sebesta, A Unified Model of Pointwise Equivalence of Procedural Computations, *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 6, November 1994, pp. 1842-1874.

[6] Holly J. Dail, "*Checkpointing and Migration in Heterogeneous, Distributed Systems*," Final Project Paper submitted for Graduate Distributed Systems Course at the University of California at San Diego, 2000.

[7] Peter. W. Smith, "*The Possibilities and Limitations of Heterogeneous Process Migration*," Ph.D. Thesis, Department of Computer Science, The University of British Columbia, October 1997.

[8] Strumpen. V and Ramkumar. B, "*Portable Checkpointing and Recovery in Heterogeneous Environments*," Technical Report 96-6-1, Department of Electrical and Computer Engineering, University of Iowa, June 1996.

[9] Balkrishna Ramkumar and Volker Strumpen, Porta-ble Checkpointing for Heterogeneous Architectures, *Proceedings of the 27th International Symposium on Fault-Tolerant Computing - Digest of Papers*, Seattle, WA, pages 58-67, June 1997.

[10] Kasidit Chanchio and Xian-He Sun, Memory Space Representation for Heterogeneous Network Process Migration, *12th International Parallel Processing Symposium*, March 1998.

[11] Kasidit Chanchio and Xian-He Sun, "*Data Collection and Restoration for Heterogeneous Process Migration*," Technical Report 97-017, Department of Computer Science, Louisiana State University, 1997.

[12] Xian-He Sun, V. K. Niak, and Kasidit Chanchio, A Coordinated Approach for Process Migration in Heterogeneous Environments, *SIAM Parallel Processing Conference*, 1999.