An Approach to Heterogeneous Process State Capture / Recovery to Achieve Minimum Performance Overhead During Normal Execution

Prashanth P. Bungale [+], Swaroop Sridhar [+]
Department of Computer Science
The Johns Hopkins University
Baltimore, MD 21218, U. S. A.
E-mail: { prash, swaroop } @ cs.jhu.edu

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 capture initiation. Current approaches incur significant performance overhead during normal execution of the process (i.e., when state capture/recovery is not being performed) in order to ensure proper initiation of state capture. This is because of their introduction of instructions into the user code, either to poll for a capture request, or to ensure correctness of self-modifying code in the case of a poll-free mechanism. In this paper, we propose a fundamentally new approach to heterogeneous process state capture and recovery that achieves minimum performance overhead during normal execution by obviating the introduction of such instructions.

In the case of high-performance computing applications, the performance gain thus achieved – especially within critical loops – would be significant. Also, our solution is suitable for effectively enabling all potential points of equivalence present in a computation if minimal latency is desired.

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 such a mechanism a key research issue in heterogeneous computing.

[+] This work was done while the authors were at the Dept. of Computer Science & Engg. of The National Institute of Engineering, Mysore, INDIA

0-7695-1926-1/03/$17.00 (C) 2003 IEEE
Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)

IEEE COMPUTER SOCIETY

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 that have *equivalent points* in all the instances of the same computation on different platforms [5].

There are many *possible* points of execution equivalence that can be identified in any program. But usually, not all of these candidates would be effectively enforced as *actual* points of execution equivalence. This is because, in order to ensure consistency, machine-dependent optimizations would need to be disabled across such *enabled* points of equivalence, which results in performance degradation. The selection of the subset of points of equivalence to be enabled 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.

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 naïve 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 overhead and latency, 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. Also, when a process state capture is requested, a breakpoint instruction to trap to the capture mechanism is placed at all "pre-emption points" (the enabled points of equivalence) available in the program to effect capture initiation. In such approaches, on architectures with varying instruction lengths, it is possible that the breakpoint instruction placed at one pre-emption point overwrites the instruction placed at another point or the current instruction (pointed to by the program counter). To avoid this loss of correctness, space has to be padded at each pre-emption point, enough to accommodate the breakpoint instruction. This can typically be done by placing dummy instructions, which introduce performance overhead during normal execution of the program.

The process introspection model proposed by Ferrari [4] and the portable (or shadow-) checkpointing model presented in [11,12] perform 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, they 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. In such an approach, a substantial amount of performance overhead would be incurred during normal execution due to continuous polling.

In applications such as process migration due to load-balancing policies, or logging mechanisms for fault tolerant systems, arbitrarily long latencies would not be acceptable. In the case of load balancing, for example, the very purpose

IEEE COMPUTER SOCIETY

of migrating the process itself is to reduce the load on the system as soon as possible. For such applications with a minimal latency requirement, almost all potential points of equivalence present in a computation must be effectively enabled. A polling approach would not be suitable because the performance overhead due to persistent polling at all such points would reach severely unacceptable levels.

4. Our Approach

4.1. Objectives

a. **Efficiency**: As a goal, in addition to providing efficient state capture and recovery, the run-time performance overhead introduced by the mechanism should be at acceptable levels. 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.

b. **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.

c. **Suitability for Minimal Latency**: The state capture mechanism should be suitable even for ensuring 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 *possible* point of equivalence in the computation.

d. **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.

4.2. Solution Overview

As described in Section 3 (Related Work), current approaches to the heterogeneous process state capture problem incur significant performance overhead during normal execution of the process. However, it is desirable that the performance during normal execution should not be degraded.

We propose a poll-free solution that involves semantics-preserving self-modification of code. However, unlike the current poll-free solutions (eg. Tui system), we do not place capture initiation instructions at all enabled points of equivalence, when a capture is requested. Instead, we place the initiation instructions in a copy of only a small portion of the code. Thus, we obviate the introduction of placeholder dummy instructions, which would have been a source of performance overhead.

The selection of the enabled subset from the entire set of potential points of equivalence shall be done according to the application-specific requirements and constraints. For example,

i) An application desiring minimal latency may effectively enable all potential points of equivalence present in a computation – something that could not be done by a polling approach, because of the exorbitant performance overhead incurred during normal execution.
ii) In the case of high-performance computing applications, only a subset of the potential points of equivalence would have to be enabled in order to exploit the performance enhancements achieved – especially within critical loops – through machine-dependent optimizations across the non-enabled, but potential, points of equivalence.

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 *enabled* 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.

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

IEEE COMPUTER SOCIETY

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 (at points which are non-reachable during normal execution but are 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 enabled points of equivalence shall not be allowed since they may prevent the different compiled versions of the program across heterogeneous platforms from hitting every such point consistently.

4.4. State Capture Initiation 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. The program counter value is used to identify the currently executing function and the current block (the segment of code between two points of equivalence containing the current instruction).
c. The following steps are carried out for ensuring that the state capture is initiated on encountering the next point of equivalence:
   i. If the program counter is exactly at a point of equivalence, an instruction to initiate state capture is placed at the same point.
   ii. If no program control instruction lies between the current point of execution and the sequentially next point of equivalence, an instruction to initiate state capture is placed at the sequentially next point of equivalence.
   iii. Else, we copy and pass control to the code fragment lying between the current point of execution and the sequentially next point of equivalence, with the following modifications made for each program control instruction lying within that fragment:
      - Code is placed in place of the program control instruction to ensure that the point of equivalence that would have been encountered next, if the program control instruction were allowed to execute, is registered.
      - An instruction to initiate state capture is then placed at the end of that code.

      Most importantly, the program control instruction is not allowed to actually execute – the steps that it would have carried out if it would have executed (for example, setting up a new activation record in the case of a procedure call instruction), will be made to be carried out, except for passing control to some point, instead of which the control is passed to the state capture mechanism.

The process is now "informed" to initiate the state capture as soon as possible. Once the process starts executing later, it eventually encounters a point of equivalence. Here, the initiation instruction (which is a call to the state capture mechanism) gets executed and the control is thus transferred to the state capture mechanism.

This algorithm requires that all points in the code which are possible destinations of jump instructions should be enabled as points of equivalence. This may not be a restrictive requirement, as optimizations would anyway not be performed across jump destinations to preserve program correctness.

4.5. State Capture Algorithm

a. Once the control is transferred to the state capture mechanism function, the following tasks are performed initially:
   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.
b. 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 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. Next, a return is performed so that control is passed to the caller's saving epilogue.
c. Step b is repeated until all activations are saved.
d. 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 [8, 9, 10]. In order to identify the data object being pointed, given a physical address, we require one of the following:

i. Compiler-support in terms of information about the positions of globals within the global data area and the

IEEE COMPUTER SOCIETY

activation record structures of the various functions, as well as run-time support only for registering heap data.

ii. Run-time support for registering local, global and heap data.

For all other data types, the data is copied verbatim into the checkpoint.

4.6 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 equivalence (at which this function activation's execution was frozen) 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. (Once all the activations have been restored, the original function entry points are restored and control is transferred to the point of equivalence at which the state capture had been initiated).
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, there is no need for any translation at all.

4.7 Performance Implications

The overhead incurred by our approach is limited to:

- the performance enhancements lost because of disallowing machine-dependent optimizations across enabled points of equivalence, and
- the run-time support required for the registration of dynamic data.

Both of these overheads are inevitable in order to preserve program correctness across heterogeneous computing platforms. Our approach thus achieves the minimum possible performance overhead during normal execution of the process.

5. Conclusion

This paper presents an approach to process state capture and recovery in heterogeneous computing systems that achieves minimum performance overhead during normal execution of the process. The solution presented, being poll-free, is suitable even for an application desiring minimal latency as it can afford to effectively enable all potential points of equivalence present in a computation. Also, high-performance computing applications can perform significantly better due to the reduced performance overhead, especially within critical loops.

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 this work.

References

[1] Ramon Lawrence, "*A 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*," in Proceedings of the Eighth International Conference on Distributed Computing Systems, pp. 104-111, 1988.

[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.

[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, Pages 1842-1874.

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

[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.

IEEE COMPUTER SOCIETY

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

[9] 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.

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

[11] Strumpen. V, 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.

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

Biographies

Prashanth P. Bungale is currently a graduate student at the Department of Computer Science of The Johns Hopkins University. He has obtained his Bachelor of Engineering degree from The National Institute of Engineering, India. His research interests are in the areas of operating systems, distributed systems and programming languages.

Swaroop Sridhar is currently a graduate student at the Department of Computer Science of The Johns Hopkins University. He has obtained his Bachelor of Engineering degree from The National Institute of Engineering, India. His research interests are in the areas of operating systems, distributed systems and programming languages.

Vinay Krishnamurthy obtained his Bachelor of Engineering degree from Vidyavardhaka College of Engineering, India. His research interests are in the areas of operating systems and distributed computing.

IEEE COMPUTER SOCIETY