Heterogeneous process state capture and recovery through Process Introspection

Adam Ferrari [a], Steve J. Chapin [b] and Andrew Grimshaw [a]

[a] *University of Virginia, Charlottesville, VA 22904, USA*
[b] *Syracuse University, Syracuse, NY 13244, USA*

The ability to capture the state of a process and later recover that state in the form of an equivalent running process is the basis for a number of important features in parallel and distributed systems. Adaptive load sharing and fault tolerance are well-known examples. Traditional state capture mechanisms have employed an external agent (such as the operating system kernel) to examine and capture process state. However, the increasing prevalence of heterogeneous cluster and "metacomputing" systems as high-performance computing platforms has prompted investigation of process-internal state capture mechanisms. Perhaps the greatest advantage of the process-internal approach is the ability to support cross-platform state capture and recovery, an important feature in heterogeneous environments. Among the perceived disadvantages of existing process-internal mechanisms are poor performance in multiple respects, and difficulty of use in terms of programmer effort. In this paper we describe a new process-internal state capture and recovery mechanism: Process Introspection. Experiences with this system indicate that the perceived disadvantages associated with process-internal mechanisms can be largely overcome, making this approach to state capture an appropriate one for cluster and metacomputing environments.

1. Introduction

The ability to capture the state of a process and later recover that state in the form of an equivalent running process is the basis for a number of important features in parallel and distributed systems. For example, process migration policies supporting adaptive load sharing and/or fault tolerance rely on a state capture facility. Process state capture and recovery is the basis of a large class of backward error recovery schemes documented in the fault tolerance literature [6]. Optimistic systems such as Time Warp [12] rely on the ability to "roll back" a local computation to provide semantic guarantees (such as the causal ordering of message delivery), and thus also require a process state capture mechanism. Distributed object systems can use process state capture and recovery to implement long-lived persistent objects efficiently, as is done in the Legion system [15].

Traditional state capture mechanisms have employed an external agent (such as the OS kernel) to examine and capture process state [19]. However, the increasing prevalence of heterogeneous cluster and "metacomputing" systems [10] as high-performance computing platforms has prompted investigation of process-internal state capture mechanisms. Among the greatest advantages of process-internal approaches are portability and the ability to support cross-platform state capture and recovery. A process can interpret the semantics of its own data regions, and thus produce a state description that can be used to reconstruct an equivalent process on a different platform. This flexibility has been found to be a valuable feature in heterogeneous cluster and metacomputing environments such as Legion, Cumulvs [14], and Dome [2].

Despite its increasing adoption in such environments, process-internal state capture and recovery has thus far been considered lacking in at least two respects: performance and usability. Performance has been considered problematic for process-internal mechanisms by a number of measures. For example, process-internal approaches require a state-capture request to be sent to the process. The delay inherent in servicing this request appears as decreased responsiveness of the mechanism in contrast to external approaches can initiate state capture at any time. Furthermore, the added overhead of maintaining meta-information (such as type) for the process's data regions can add run-time overhead, lowering raw computational performance. Lack of usability of process-internal mechanisms is due mainly to added programmer effort and/or limitations on the types of services the process can use. For example, in some cases, such as Legion, the code needed to describe and recover the process's state must be provided by the user.

In this paper we describe a new process-internal state capture and recovery mechanism: Process Introspection. This system is based on a combination of library and compiler support to maximize the ease of use of process-internal state capture and recovery. For platform-independent modules, the compiler completely automates state capture and recovery. For modules where automatic transformation is not possible, a flexible library providing the needed primitive operations for cross-platform state capture and recovery makes adding state capture functionality straightforward. In section 2 we describe the design and interface of our system, and in section 3 we present key features of the implementation. We argue that the design of the Process Introspection system overcomes the usability flaws in existing process-internal state capture mechanisms. In section 4 we describe the results of performance measurements of our system. These results indicate that a cross-platform process-internal state capture mechanism can offer good performance. Our results lead us to conclude that the process-internal approach to state capture is the appropriate one for cluster and metacomputing environments. In section 5 we describe related work, and in section 6 we discuss our conclusions.

2. Design

The design of a process-internal state capture mechanism is naturally based upon the modification of user programs to render them both self-describing and self-recovering. Beyond the basic goal of providing a platform-independent state capture and recovery service, important goals for such a mechanism are to provide good performance and ease of use. In this section, we describe the key features of our design as a basis for discussing how it meets these goals.

2.1. Model

In our model, a running process is defined to be in one of three states: *normal execution*, *state-capture*, or *state-recovery*. The state of the process is changed by the program itself, either in response to requests from outside sources or as the result of an internal trigger such as periodic checkpoint scheduling. We require that the program periodically execute *poll points*: points in the code at which the process determines if it is in the *state-capture* mode, in which case a state description should be produced if one has been requested (analogous to Bus Stops in Heterogeneous Emerald [23]). Certain parts of the process state are easily captured – for example, any global variable or heap allocated data structures are globally addressable and are thus easy to manipulate. The key difficulty in creating the state description is the capture of the subroutine invocation stack state.

In the Process Introspection approach, the process utilizes the native "subroutine return" mechanism to capture stack state. When a poll point is encountered during *state-capture* mode, the current active subroutine captures its own state, including its local variables and the logical location of the poll point at which the current call frame was saved (e.g., this can simply be an integer that uniquely identifies the poll point within the subroutine), and returns to the caller. After the return, the caller saves its own state in the same way, and this frame-by-frame stack capture repeats until the base subroutine has been reached, at which point the stack state capture is complete. For this stack-saving mechanism to work, the program must execute a poll point after returning from each subroutine call. At this point, the program might be in *normal execution* mode, in which case it proceeds with normal processing, or it might be in *state-capture mode*, in which case the stack save process continues. We name these required poll points following subroutine returns *mandatory poll points*. In fact, more frequent checks for state capture initiation may be desirable, in which case additional *optional poll points* can be placed anywhere in a program.

A side-effect of capturing state in this manner (but not of simply polling) is the destruction of all data on the call stack, implying that the program must perform some of the work of a restart to recover the stack if it is to continue execution after capturing state. We note an important optimization to the above mechanism: in the process of capturing the state of each frame, the frame state should also be saved in memory. This permits the implementation of a quick stack recovery after state capture. Because the state capture mechanism is non-destructive to other state (i.e., global variables and dynamically-allocated memory blocks), this optimization permits the process to proceed without unreasonable delay after state capture.

A further optimization to the described code modification scheme is also possible. Although we initially stated that mandatory poll points must be placed immediately following every subroutine call statement, we can in practice loosen this restriction. Given knowledge that all possible call chains resulting from a subroutine call would contain no poll points, the mandatory poll point following the call site can safely be omitted. For example, consider a call to a simple function that calls no other functions and contains no poll points. Upon return from this function, we know that a capture of the stack could not yet have been initiated. Even if state capture had been requested while the function was executing, we can safely continue normal execution after the call returns before beginning to service the request.

To effect restarts, the process employs the native "subroutine call" mechanism. On a restart, the program is started and is immediately placed in *state-recovery mode*. The base subroutine of the program always executes a prologue that checks for *state-recovery* mode, then conditionally recovers the data for its local stack frame and jumps to the location in the subroutine for the call to the next stack frame, i.e., the poll point at which the state of the current stack frame was captured. Each stack frame in turn is recovered by its respective subroutine, which must implement its own stack-recovery prologue. Before jumping just past the poll point that initiated the state capture, the final frame resets the process state to *normal-execution* mode to complete the state recovery and resume normal execution. Of course, at some point during the recovery process, the global variables and heap allocated data must also be restored.

With these additions, the program can restore an intermediate state as produced by its own state capture mechanism. In particular, since the state capture and recovery mechanisms are specified at a platform-independent level of representation, different implementations of the program (i.e., versions compiled for different architectures) can read and write one another's captured state, assuming the associated data is stored in a universally recognizable format, masking issues such as data representation (cf. Sun XDR [21]).

2.2. System usage

This model of process-internal state capture and recovery appears at first glance to require significant programmer effort. In practice, many of the described code transformations can be automated. The Process Introspection system does this through the use of a source code compiler and runtime support library. For computational modules that are specified in a platform-independent form (i.e., that are written in a high-level language, are type-safe, and do not rely on the underlying features of a particular hardware platform for correctness), the described code transformations can be completely automated.

We expect this completely automatic usage of the system to be the typical mode employed by application programmers such as domain scientists. However, some program modules are not amenable to automatic transformation. For example, modules that deal with state that is external to the process (e.g., file system interfaces, communication interfaces) cannot be automatically transformed by our system to incorporate state capture and recovery. In some cases, it seems that this is in fact an inherent limitation of our approach. Consider a message passing interface module. A compiler attempting to incorporate state capture and recovery functionality into such a module would have no way to know how to encode the capture of state such as messages in transit, nor would it be able to determine the state capture coordination semantics required by the application. In such cases, our model requires that the module be augmented by hand to incorporate state capture and recovery functionality – this typically involves the creation of a state-capture-enabled wrapper module for the interface in question. We envision the creation of state-capture-enabled library modules as an infrequent activity undertaken by cluster and metacomputing software system designers. These wrapper libraries can then be reused by application programmers whose own modules will be transformed automatically. To support the interoperation of state-capture-enabled library modules and automatically transformed application modules, and to ease the hand-coding of state capture mechanisms for library modules, our system supports a library interface. This library provides basic services such as cross-platform data-format transformation routines, routines for constructing descriptive meta-information about the data regions of the process (such as a data type description table), and an event model for allowing separately developed state-capture-enabled modules to interoperate.

3. Implementation

3.1. Overview

We have constructed a prototype implementation of the Process Introspection system consisting of a library module as described in section 2.2, the Process Introspection Library (PIL), and a source code translator called APrIL (Automatic application of the Process Introspection Library) which can automatically apply the Process Introspection transformations to platform-independent modules written in ANSI C. The implementation has been tested on a variety of workstations and PC platforms, including Sun workstations running Solaris or SunOS 4.x, SGI workstations running IRIX 5.x and 6.x, IBM RS/6000 workstations running AIX, DEC Alpha workstations running Digital Unix or Linux, and PC compatibles running Linux or Microsoft Windows 95/NT.

3.2. The Process Introspection Library

The Process Introspection Library (PIL) is the most basic component of the system. In the case of hand-coded modules, the PIL provides the API for implementing a module's state capture and recovery capability. In the case of compiler-transformed modules, the PIL provides needed run-time support. The primary job of the Process Introspection Library is to provide an easy-to-use mechanism for describing, saving, and restoring data regions in as automatic a fashion as possible. In addition, the library provides an event-based mechanism for coordinating the activities of modules during state capture and recovery.

The key elements of the PIL are:

The Type Table: To capture or restore a memory block, the PIL must have a description of the basic data types stored in that memory block. The PIL provides an interface to a table which maps type identifiers to logical type descriptions. These type identifiers can then be used to tag data regions, indicating the types of the data found in the region. The type table is not unlike a type description table that might be found in a compiler, except that it is available and dynamically configurable at runtime. The interface provides pre-defined type identifiers for the basic types supported by ANSI C, and provides an interface for composing vector and structure descriptions based on existing types.

Data Format Conversion Module: The PIL provides an interface for reading and writing typed data from and to a state description in an architecture-independent format, respectively. This interface is responsible for masking differences in byte ordering and floating point representation. When storing captured state, the library automatically includes a description of the data formats used. Later, during state recovery, the data format can be converted to the restarting processor's representation, a protocol known as *receiver-makes-right* [26]. Given this approach, the library must contain routines to translate the set of basic data types from every available format to every other available format. This O(n^2) requirement (where n is the number of different data formats) may initially appear unnecessarily costly; why not instead use an O(n) solution such as XDR? In fact, the receiver-makes-right protocol makes sense only in light of the small number of data formats actually used by current computer systems and because the cost of format conversions is avoided for the frequent case in which captured state is recovered on a computer with similar data formats to the one on which it was created.

Pointer Analysis Module: Memory addresses (i.e., pointers) contained within memory blocks are inherently platform-dependent. Thus, they must be stored using a logical format in place of the physical address. Similarly, at state recovery time, logical pointer descriptions must be translated to determine the actual local memory address values that should be restored. Our mechanism for this assigns a unique identification number to every memory block of interest in the program, and a logical pointer description comprises a <memory block identification tag, offset> tuple. Based on this idea, the Pointer Analysis Module provides a convenient interface for generating logical descriptions of memory locations and for mapping these logical descriptions into actual memory addresses. The pointer description implementation is based on simple case analysis; a pointer can be one of exactly five types: a reference into a heap allocated memory region, a reference into a global memory block, a reference into a local (stack) memory block, a pointer to some code entry point, or value which has special meaning in the program (such as NULL in C). The PIL associates id numbers with memory blocks of each class (except the last), providing the basis for logical pointer description.

The primary challenge posed by the use of logical pointer descriptions is the translation of these descriptions during state recovery – offsets into a memory region may need to be transformed due to differences in data size and alignment between the capture and recovery platforms. The Pointer Analysis Module uses a pointer translation algorithm that transforms offsets based on the type descriptions available for all memory regions (supported by the Type Table), and knowledge of the data formats and alignment issues of the source and target platforms. Full details of this algorithm are presented in [7].

Global Variable Table: The PIL provides an automated mechanism for saving and restoring the values of global variables at state capture and restart time, respectively. The mechanism requires that the memory addresses, type table indices, and sizes of all globally-addressable memory blocks be registered with the PIL in a Global Variable Table. Besides providing automatic capture and recovery of globals, the Global Variable Table is used to perform pointer analysis for addresses pointing within global memory blocks.

Heap Allocation Module: The PIL provides a mechanism for allocating memory blocks from the heap that will be automatically captured and restored. The Heap Allocation Module exports heap wrapper routines that perform typed memory block allocation and deallocation. Similar to the case with globals, these wrapper routines maintain a table of the addresses, type table indices, and sizes of all active dynamically-allocated memory blocks.

Code Location Table: To fully resolve the meaning of all pointers, the PIL must maintain a table that maps logical code entry points to actual memory code locations. All subroutine entry points (and other addressable code locations) in a program that may be referred to by a pointer must be assigned a logical identification number via the Code Location Table interface.

Active Local Variable Table: Because pointers can refer to local variables, the addresses, type table indices, and sizes of some local variable memory blocks should be registered with the PIL. Note, only those locals whose addresses are ever examined (and thus whose addresses might consequently be found in some memory block) need to be registered with the Active Local Variable Table. Local variables whose addresses are never examined should not be registered, preserving the possibility of register assignment.

Event Module: The Event Module provides the primary mechanism for modules to customize their state capture and recovery behavior. The Event Module allows a program module to register function callbacks that will be invoked by the system automatically at state capture and recovery time. To understand the importance of this module, consider the case of a file system interface module. Besides the normal activities of saving and restoring the data in memory blocks (as is done by every module, and which is typically automated using the PIL), the file module must perform extra actions. During state recovery, for example, it must use the local file interface to re-open the files that were in use when state was captured. It might also be responsible for maintaining the file version differences associated with different captured states. These extra activities can be coded in the form of event handlers which would be executed in response to state capture and restart events.

3.3. The APrIL source code translator

The programming interface provided by the PIL automates some of the elements of the Process Introspection model, but is still relatively low-level. Although issues such as data representation are handled, using only the PIL the programmer would be left to manually perform code modifications such as poll-point placement and prologue generation. Fortunately, a source code translator can automate this process for platform-independent programs. Using the Sage++ toolkit [3], we have implemented this functionality in the APrIL compiler. APrIL takes as input ANSI C code, and produces as output new ANSI C code transformed to utilize the PIL as a run-time interface. The resulting C code can then be compiled using any ANSI C compiler. In this section, we examine the fundamental APrIL transformations: poll points, function prologues, and function epilogues.

```
_PIL_PollPt_1:
      if (PIL_State & PIL_CaptureNow) {
            PIL_PushCodeLocation(1);
            PIL_State |= PIL_CaptureInProgress;
            goto _PIL_save_frame_;
      }
```

Figure 1. An optional poll point.

3.3.1. Poll points

APrIL inserts poll points throughout the code it transforms. At each poll point, code is inserted to check if the process is in state capture mode – this simply involves examining the value of the global variable PIL_State (we currently set the value of this variable using an interrupt handler that processes external state capture requests in the form of signals; in principle it could be set by other triggers such as periodic checkpoint scheduling, or a state capture request message). Immediately following the poll point, code is inserted which will be executed when state capture is in progress. This code records the location in the current subroutine at which state is captured and jumps to a function epilogue that saves the actual parameters and local variables in the frame.

As described in the model, APrIL generates two kinds of poll points: optional and mandatory function call site poll points. Optional poll points can be inserted in the transformed code between any two statements in the universal representation. These poll points are designated by a single labeled code location (i.e., a C label statement). An example of an optional poll point is depicted in figure 1.

In our model, mandatory poll points are inserted by APrIL after every function call statement in the code[1] – these mandatory poll points are required to implement the stack save mechanism based on the native function return mechanism. When a function returns in APrIL-transformed code, the return may be due to the normal completion of the function, or it may be a return being performed in the context of capturing the stack. Mandatory poll points must catch and implement this latter case. Mandatory poll points require two labeled code locations: one before the call site (to handle the case that state capture was initiated in a higher call frame), and one after the call site (in the event that state capture is initiated immediately following a normal function return). An example of a mandatory poll point is given in figure 2. Note that if this poll point continues

[1] In fact, function calls in C occur not as specific statements, but instead within expressions. Expressions in turn can appear within other expressions, in more complex statements, etc. (e.g., a function call might be a parameter to another function call, which might be part of the conditional for an “if” statement). To perform the transformations as described in the model, APrIL utilizes a pre-processing step in which it extracts functions from complex expressions and statements, and reduces them to simple C expression statements containing a single function call.

```
_PIL_PollPt_2:
      i = function(A,X,100);
_PIL_PollPt_3:
      if (PIL_State & PIL_CaptureNow) {
            if (PIL_State & PIL_CaptureInProgress) {
                  PIL_PushCodeLocation(2);
            } else {
                  PIL_PushCodeLocation(3);
                  PIL_State |= PIL_CaptureInProgress;
            }
            goto _PIL_save_frame_;
      }
```

Figure 2. A mandatory poll point.

a stack capture that was initiated in a higher call frame, the code location is recorded as 2 to ensure that on recovery this frame will jump to label _PIL_PollPt_2, which will result in a call to the next function needing to restore state. On the other hand, if this poll point initiates state capture, it records the code location as 3 to ensure that on recovery the frame jumps beyond the function call – because this poll point initiated the stack capture, the call to function must have completed normally (without capturing state), and thus we must not re-call the completed function on recovery.

The placement of poll points in the code is a critical performance issue for APrIL. If poll points are placed so that they occur frequently, the introduced overhead may be large. On the other hand, if poll points are placed too infrequently, a state capture request sent to the process may suffer a long delay before being serviced. Clearly, a balanced approach based on the user’s tolerance of introduced overhead and state-capture-request wait time is required. If the user expects to perform state capture operations infrequently (e.g., once every minute), but demands little introduced overhead, then very sparse, conservative poll-point placement is called for. Alternatively, if state-capture-request wait times must be very low (for example, if state capture will be used for code migration to effect load sharing), more frequent, aggressive placement is appropriate. However, the problem of statically examining code and determining the introduced overhead and resulting state-capture-request wait time based on a given poll-point-placement strategy is difficult, if not impossible. The current APrIL solution is to provide a set of heuristic placement strategies with varying degrees of placement aggressiveness.

Our currently-supported placement strategies are based on the observation that the primary mechanisms for induction in procedural programming are iteration (loops) and subroutine invocation (recursion). Although subroutine invocation already causes periodic polling (due to mandatory poll point placement), it seemed likely that the addition of optional poll points into loops could provide more complete periodic polling coverage over the lifetime of a program and thus lead to lower on-average state-capture-request wait times. Care must be taken, however, as the naive policy of placing a poll point inside each loop body could lead to poor performance – careless placement of poll points can prevent the application of many back-end optimizations, including those performed on loops. To allow further control over the placement of poll points, the APrIL heuristic policies classify loops based on the number of statements in the loop body and the nesting level of the loop. Compile time switches allow the user to restrict placement to loops with given characteristics. Examples of possible policy selections include:

- No optional poll points (mandatory poll points only).
- Optional poll points placed as the last statement in each nesting loop (i.e., each loop that contains at least one other loop).
- Same as the previous, but poll points are added only to outermost loops with greater than k statements.

Although many more policies are available, a better interface to the compiler would allow the programmer to specify the desired performance characteristics of the transformed code, which would guide the automatic selection of a policy. The degree to which this ideal can be approximated is the subject of future work. The performance characteristics of three currently available APrIL placement options are examined in section 4.

3.4. Function prologues

Function prologues are added to every function definition transformed by APrIL. If the addresses of any local variables or parameters (i.e., any objects stored on the stack) are examined in the function body, APrIL generates calls to the PIL to register those variables in the local variable table. APrIL then generates a check to determine if the process is in state recovery mode (recall that stack restoration in our model is implemented using the normal function call mechanism). APrIL generates code to be executed in case of a restart, which will restore the values of all local variables and actual parameters (using the PIL interface), determine the code location in the function at which the state for this frame was captured, and jump to the appropriate poll point label in the function. Figure 3 illustrates an APrIL function prologue transformation. The function heading given in figure 3(a) is transformed to include the prologue depicted in figure 3(b).

This function has an array X whose address is used at some point in the function, and thus a call to register the address, size, and type of this array is generated. The prologue checks the value of the `PIL_State` variable to determine if this function call was made in the process of restoring a call stack. If it was, the actual parameters and locals are restored using PIL routines. The point in the function at which the state was captured is then jumped to using a `goto` based on a code location marker read from the captured state. For some code locations (those corresponding to optional poll points and the second label associated with mandatory poll points), the generated code first calls `PIL_DoneRestart()` to complete the restart process and unset the `PIL_State` variable.

```
void example(double *A) {
      int i;
      double X[100];
```

(a) The original function heading.

```
void example(double *A) {
      int i;
      double X[100];
      PIL_RegisterStackPointer(X,PIL_Double,100);
      if (PIL_State & PIL_RecoverNow) {
            int PIL_code_loc;
            A = PIL_RestoreStackPointer();
            i = PIL_RestoreStackInt();
            PIL_RestoreStackDoubles(X,100);
            PIL_code_loc = PIL_PopCodeLocation();
            switch(PIL_code_loc) {
                  case 1:     PIL_DoneRestart();
                              goto _PIL_PollPt_1;
                  case 2:     goto _PIL_PollPt_2;
                  case 3:     PIL_DoneRestart();
                              goto _PIL_PollPt_3;
            }
      }
```

(b) The transformed function heading.

Figure 3. A function prologue transformation.

3.4.1. Function epilogues

Poll points inserted by APrIL generate code to jump to a function epilogue during state capture to save all of the local variables and actual parameters for the function. APrIL generates an epilogue for each function it transforms that contains any poll points (if the function never polls for state capture requests, it will never need to save its state) placed beyond the last return statement; the epilogue is accessible only by `goto`, and is not executed during the normal progression of the program. The function epilogue for the example function from figure 3 is depicted in figure 4.

This design for saving the local state associated with a function call has the inherent implication that all local variables must be visible from the outermost scope of the function. To ensure this, APrIL moves the declaration of locals declared in inner scopes to the head of the function, renaming where appropriate to avoid name clashes.

```
_PIL_save_frame_:
        PIL_SaveStackPointer(A);
        PIL_SaveStackInt(i);
        PIL_SaveStackDoubles(X,100);
        return;
```

Figure 4. A function epilogue.

3.4.2. Module initialization

The three types of transformations discussed thus far are primarily aimed at implementing the state capture and recovery of function call stacks. APrIL also generates a routine to register any types defined by the translated module with the Type Table and register any globals defined by the translated module in the Global Variable Tables. The generation of this function is a straightforward process based on any types and global variables found in the module.

3.4.3. Heap allocation transformations

One of the more difficult transformations that APrIL performs is the translation of all heap allocation requests into calls to the typed allocation routines provided as part of the PIL. Since heap allocation is not part of the C language syntax but is instead handled by library routines, APrIL is required to perform a heuristic to determine when heap allocation is taking place, and the type and size of the allocated memory. The currently implemented heuristic finds all calls to the standard C library heap allocation routines (e.g., `malloc()`, `calloc()`, `realloc()`, etc.), uses the parameters to the call to determine the allocation size, and attempts to determine the allocation type first based on the type that the return value is cast to (if it is available), and (failing that), on the type of the variable to which the return value is assigned. While this heuristic is adequate in many cases, it can fail if the memory allocation method used does not match our expected patterns. Future work will include investigating better heuristics for finding and wrapping heap allocations.

4. Performance

To examine the performance characteristics of our prototype implementation, we applied the system to a set of numerical applications. Tests were run on the heterogeneous set of test platforms listed in table 1. This set of test programs included:

- mm (matrix multiply) – Computes the product of two dense, square matrices of 256×256 double precision floating point numbers using the standard $O(n^3)$ algorithm.
- gs (Gauss–Seidel) – Solves the sparse linear system of 10^4 equations resulting from the discretization of a two dimensional Poisson equation with Dirichlet boundary conditions. The algorithm used is a standard Gauss–Seidel five point stencil iteration applied to a 80×80 grid of solution elements until the change in the two-norm of the solution is less than 10^{-2}.
- qs (quicksort) – Applies a standard quicksort algorithm to an array of 2^{21} integers.
- ge (Gaussian elimination) – Performs Gaussian elimination with partial pivoting on a dense 512×512 matrix, followed by a back-substitution phase to obtain the solution vector.
- cg (conjugate-gradient) – Applies a basic conjugate-gradient iteration (no preconditioning) to the same linear system solved by the Gauss–Seidel test, using the same convergence criterion as that example, with the solution discretized onto a 200×200 grid.

Table 1
Test platforms.

Platform		RAM	OS	Compiler
x86	200 MHz Pentium Pro dual processor	64 MB	Linux 2.0	GNU gcc 2.7.2
Alpha	500 MHz DEC Alpha	128 MB	Linux 2.0	GNU gcc 2.7.2
RS/6000	PowerPC 601-based IBM RS/6000	128 MB	AIX 4.2	xlc 3.1
MIPS	100 MHz MIPS R4000	64 MB	IRIX 6.2	SGI cc
SPARC	50 MHz 4-processor SparcStation-20/514	512 MB	SunOS 5.5.1	SPARCCompiler C 3.0

Our first set of measurements was performed to examine the run-time overhead introduced by our code transformations. The transformations applied by APrIL add overhead to programs not only because they result in the execution of extra instructions, but also because they affect the ability of compilers to apply certain optimizations. The degree to which the APrIL code transformations affect performance is primarily a function of two factors: the policy for placing poll points in the code, and the characteristics of the code itself. To examine the effects of these factors, we applied three of the available set of heuristic poll-point-placement policies supported by APrIL to each of our test applications. The selected transformation heuristics were:

- Mandatory – no optional poll points placed, only those required for correct capture and recovery of the subroutine invocation stack.
- Conservative – in addition to mandatory poll points, this policy places an optional poll point after the last statement in the body of each nesting loop (i.e., a loop containing at least one other loop in its body).
- Aggressive – in addition to those placed by the conservative policy, this policy places a poll point in each loop with more than one statement in its body. For example, whereas the conservative policy would not place a poll point in an innermost loop, this policy may perform such a placement.

We timed each of the test programs, first compiled without APrIL transformations, and then transformed using each

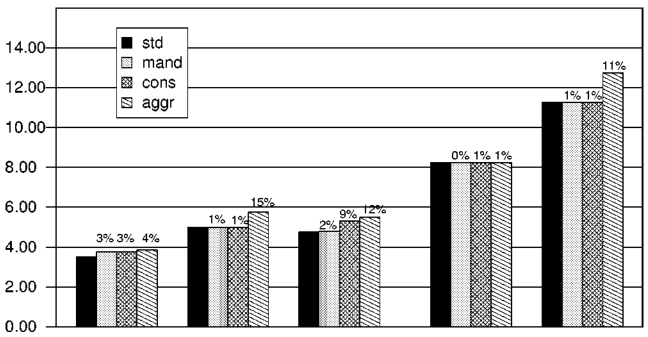

Figure 5. Execution time of optimized example programs (std) compared to execution time of optimized transformed programs (mand, cons, aggr) on x86 platform. Times in seconds. Percent increase in run time is indicated for each transformed program.

of the above policies ("-O" optimization was performed in all cases). For the transformed versions, we measured the time to completion without capturing or recovering during execution (measured run times included time to load the process). In figure 5 we display the relative performance of non-transformed and transformed test programs on the x86 platform. Additional results for the other test platforms are presented in [7] – because all platforms exhibited roughly the same relative performance for the different transformed program versions, we present the timings for only one architecture here.

The first trend that we observe in these results is that the overhead of our transformations under the mandatory and conservative placement policies is generally low – well below 10% in all cases except for quicksort under the conservative policy. As expected, more aggressive placement can easily lead to high overhead. However, we note that the impact of the poll-point-placement policy is dependent on the application. Given our loop-structure based placement heuristics, the loop structure of an application will largely determine the performance implications of a given policy. Finally, we note that in all cases, at least one of the examined policies was able to achieve low net overhead.

Low introduced overhead, while important, is only half of the story. Recall, the frequency at which poll points are encountered not only affects overhead, but also determines the average amount of time that state capture requests would have to wait before being serviced. In our model, a state capture request is sent to the process, resulting in a global variable being set to indicate that state capture has been requested. It is only later, when the process reaches a poll point, that the state capture actually begins. This naturally leads to the question, if a process is sent a state capture request, how long will it be before a poll point is reached and state capture begins? To investigate the state-capture-request wait time resulting from our system, we modified the APrIL compiler to instrument transformed programs at each poll point to keep a running count of the number of poll points encountered. Assuming an approximately equal distribution of poll points encountered over time, the average interval between poll points is simply the execution time divided by the poll point count. Based on measurements of the poll point counts, we present the computed average poll point interval for each program on the x86 test platform in table 2.

Table 2
Average poll point interval (on x86 platform). Times in milliseconds unless otherwise noted.

Policy	mm	gs	qs	ge	cg
Mandatory	3.7 sec.	0.42966	0.00026	1.2 sec.	5.16433
Conservative	0.05606	0.01057	0.00017	0.06234	0.20863
Aggressive	0.02769	0.00015	0.00016	0.06175	0.00022

These results put the performance overheads presented in figure 5 in perspective. First, we note the correspondence between high poll point counts and high introduced overhead. This confirms our intuition that introduced overhead will be a function of the frequency at which poll points are encountered. Next, we note that optional poll point placement is important. For the matrix multiply and Gaussian elimination examples, the mandatory-only policy resulted in very few poll points, and thus very high average poll point intervals. This result is an artifact of the structure of these applications: each uses few function calls, and coarse, long-running loops – attributes that are not uncommon in high performance applications. Thus, we conclude that mandatory placement alone is insufficient for some applications.

Perhaps the most important result these measurements provide is that for each application, at least one of the examined policies resulted in both low introduced overhead (i.e., below 10%) and a small average poll point interval – generally below 0.1 millisecond. Since captured state will generally be written to stable storage (e.g., checkpoint/restart applications) or over a network (e.g. migration applications), poll point intervals of this duration are orders of magnitude less than the time required to perform state capture. Thus, for all applications, an acceptable level of overhead and very low on-average poll point interval were both possible with at least one policy. The implication of this fact is that Process Introspection can be applied both automatically, and with good performance.

Our final set of experiments was performed to examine the efficiency of the state capture and recovery mechanisms. For these tests, we instrumented the PIL to note the time when either state capture or recovery was initiated, and to subsequently record the completion time. To obtain repeatable results, we also instrumented the PIL to automatically force a state capture request after a set number of poll points encountered during execution. We ran each of our transformed test applications (compiled with the conservative poll point placement policy) until 50000 poll points were encountered. At that point, a checkpoint was written to disk, and the process was terminated. We then used the checkpoint files produced on each platform to time a restart from disk. In figure 6 we present the results for the x86 platform.

We found that state capture and recovery costs on each platform were generally a function of state size and the

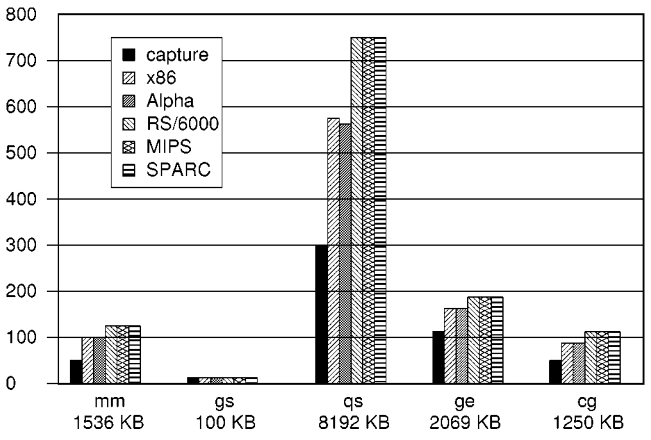

Figure 6. State capture and recovery costs on x86 platform. Times in milliseconds; recovery time listed for checkpoints produced on each of the five test platforms. The captured state size for each test program is listed under the results for that program.

I/O performance possible on the test platform. For example, in figure 6 we note that the time to capture the state of the Gaussian elimination program is roughly twice that required to capture the state of the conjugate gradient program, and there is approximately a factor of 2 difference in the programs' state sizes. An important result we notice is that the cost of restarting from a checkpoint produced on a platform with an incompatible data representation is greater than that of a compatible restart. For example, the times to perform state recovery from the incompatible RS/6000 format presented in figure 6 are generally about 40% greater than the times required to restart using the native x86 format. The RS/6000 uses big-endian byte ordering, and thus extra time is required during state recovery to perform byte swapping. This result is not only important for making scheduling decisions (e.g., this might affect selection of a target host for migration), but also confirms our intuition that receiver-makes-right data conversion can improve performance.

5. Related work

The idea of capturing the state of a running process on one kind of computer system and then later restarting an equivalent process on a different type of computer system has been the subject of a number of previous papers. Perhaps the most general coverage of this topic is presented by von Bank et al. in [25]. In this paper, the authors identify the general idea that a procedural computation can be modeled as progression through a sequence of compatible well-defined states: points in execution at which the state of a process can be used to fully describe the equivalent state of any other implementation of the process. In our model, these compatible well defined states are present in the form of process states when poll points are encountered. Related implementation work done by this group integrated a limited form of heterogeneous process migration into the V system [5]. As is typical in existing approaches, this implementation relied on the operating system to examine and translate the state of the process.

A novel approach to the heterogeneous state capture/restore problem was proposed by Theimer and Hayes [24]. In their proposed solution, the state of a process is examined and captured using compiler-generated symbol mapping information. Instead of being captured in a data-only format that must be used in conjunction with a separate executable (a feature common in the other systems presented in this section, as well as our own), the process state is instead captured in the form of an intermediate code program. This program is constructed to re-initialize the full equivalent state of the captured process and proceed from its logical point of state capture. The actual process migration then consists of compiling this program on the destination machine. Such a mechanism would have the desirable property of requiring very little external support at the restart host (beyond the ability to recompile the intermediate code program). Our approach extends this desirable feature of autonomy to include state capture as well as state restore.

A more recent and fully implemented approach to the heterogeneous state capture problem was presented by Steensgaard and Jul in [23]. In this paper, the authors describe an extension of the thread- and object-mobility capability of the heterogeneous Emerald distributed system to allow native code migration among heterogeneous hosts (previous implementations supported native code mobility for homogeneous hosts). In their implementation, native code threads can migrate at well-defined points during execution, called Bus Stops, at which time control is transferred to the Emerald run-time system, and a complete description of the running code is constructed by the system using compiler-generated mapping information (the same principle as used for symbolic debugging). This approach has the attractive property that modification to the generated code is not required; the compiler is simply responsible for generating the extra mapping information required by the run-time system. This approach differs from ours in exactly this respect – while we require modification of programs to support state capture and recovery, we do not require support from any external agent for this functionality. This affords us the desirable attribute of generality – our tool can be integrated into existing distributed systems without requiring modification to those systems or to our basic process state capture mechanism, and Process Introspection does not require extensive run-time system support. Our current implementation requires only that the system interface be accessible from C code, and that it be possible to construct a wrapper interface for system services that maintain external state for processes.

A similar approach to that of heterogeneous Emerald called Tui [20] has been proposed by Smith and Hutchinson. This approach also involves the use of compiler-generated state mapping information in the form of the symbol table typically used by symbolic debuggers. The Tui implementation has the additional desirable feature of supporting programs written in C. Again, this approach differs from Process Introspection in being external-agent-based – special programs are required to capture and restore the process state.

Recent work in the area of mobile agents has resulted in a number of state-capture and recovery mechanisms to support migration in mobile agent languages. For example, the Sumatra [1] language supports the capture and recovery of Java threads in a heterogeneous environment. State capture and recovery in Sumatra is achieved through modifications to the Java Virtual Machine bytecode interpreter. A more flexible approach is supported by the Ara system [17]. As opposed to Sumatra, which mandates use of the Java language, Ara supports mobile agents in an extensible set of interpreted languages, currently including interpreted C and Tcl. To support state capture of a running agent, the interpreters used in the system must be able to capture their own full state (i.e., including the state of a program being interpreted). A primary drawback of these and many other mobile agent systems is the use of interpreted execution for agents. In our intended application domain, this model fails to meet the performance requirements of most users. A notable system that overcomes this limitation is Extended Facile [13], an agent programming system based on the Facile functional programming language. In Extended Facile, agents are first-class functions which may be transferred to remote nodes for execution. The code for agent functions in Extended Facile can be transferred in a higher-level, platform independent representation or as native-code executable instructions (or a mixture of the two). Extended Facile utilizes the continuation-based compilation model of the language to support state capture and recovery at function boundaries.

6. Conclusions

We have presented Process Introspection, a process-internal heterogeneous process state capture and recovery mechanism based on automatic code modification. Experiences with this system have produced encouraging results. First, we found that relatively simple poll-point-placement policies can achieve acceptable levels of incurred overhead while at the same time providing good performance in terms of average checkpoint-request wait time. This result is important – process internal state capture and recovery made possible by periodic polling can be utilized effectively, efficiently. Furthermore, the design of our system demonstrates that process-internal state capture and recovery need not place undue burden on the programmer – the typical usage mode for our system is fully automatic, requiring only an additional compiler translation of the user's application program.

We believe our mechanism is general and widely applicable in a variety of different distributed system environments. For example, we are currently working on adapting the system for use in the Legion [15] metacomputing system, and are investigating integration into a PVM [9] or MPI [11] system. This adaptability is explicitly supported by our PIL API which provides a medium for APrIL-transformed modules and hand-coded system-interface wrapper modules to interoperate. Furthermore, we have designed extensions for our system to handle additional programming constructs such as threads, and languages such as Fortran and C++. These designs (presented in [7]) are the subject of ongoing development and evaluation.

References

[1] A. Acharya, M. Ranganathan and J. Saltz, Sumatra: A language for resource-aware mobile programs, in: *Mobile Object Systems*, eds. J. Vitek and C. Tschudin (Springer, Berlin, 1997).

[2] A. Beguelin, E. Seligman and M. Starkey, Dome: Distributed object migration environment, Technical Report CMU-CS-94-153, Carnegie Mellon University (May 1994).

[3] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas and B. Winnicka, Sage++: An object-oriented toolkit and class library for building Fortran and C++ restructuring tools, OONSKI (1994).

[4] J. Casas, D.L. Clark, P.S. Galbiati, R. Konuru, S.W. Otto, R.M. Prouty and J. Walpole, MIST: PVM with transparent migration and checkpointing, in: *3rd Annual PVM Users' Group Meeting*, Pittsburgh, PA (May 7–9, 1995).

[5] F.B. Dubach, R.M. Rutherford and C.M. Shub, Process-originated migration in a heterogeneous environment, in: *Proceedings of the ACM Computer Science Conference* (February 1989) pp. 98–102.

[6] E.N. Elnozahy, D.B. Johnson and Y.M. Wang, A survey of rollback-recovery protocols in message-passing systems, Technical Report CMU-CS-96-181, Carnegie Mellon University (October 1996).

[7] A.J. Ferrari, Process state capture and recovery in high-performance heterogeneous distributed systems, Ph.D. thesis 9802, Department of Computer Science, University of Virginia (January 1998).

[8] R.F. Freund and D.S. Cornwell, Superconcurrency: A form of distributed heterogeneous supercomputing, Supercomputing Review 3 (October 1990) 47–50.

[9] A. Geist, A Beguelin, J. Dongarra, W. Jiang, R. Manchek and V.S. Sunderam, *PVM: Parallel Virtual Machine* (MIT Press, Cambridge, MA, 1994).

[10] A.S. Grimshaw, J.B.Weissman, E.A. West and E. Loyot, Meta systems: An approach combining parallel processing and heterogeneous distributed computing systems, Journal of Parallel and Distributed Computing 21(3) (June 1994) 257–270.

[11] W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (MIT Press, Cambridge, MA, 1994).

[12] D.R. Jefferson, Virtual time, ACM Transactions on Programming Languages and Systems 7(3) (July 1985) 404–425.

[13] F.C. Knabe, Language support for mobile agents, Ph.D. thesis, available as Technical Report CMU-CS-95-223, Carnegie Mellon University (December 1995).

[14] J.A. Kohl and P.M. Papadopoulos, Efficient and flexible fault tolerance and migration of scientific simulations using CUMULVS, in: *2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, Welches, OR (August 1998).

[15] M.J. Lewis and A.S. Grimshaw, The core Legion object model, in: *Proceedings of IEEE High Performance Distributed Computing 5*, Syracuse, NY (August 6–9, 1996) pp. 551–561.

[16] 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* (1988) pp. 104–111.

[17] H. Peine and T. Stolpmann, The architecture of the Ara platform for mobile agents, in: *Proceedings of the First International Workshop*

on Mobile Agents: MA'97, Berlin, Germany (April 7–8, 1997), eds. K. Rothermel and R. Popescu-Zeletin, Lecture Notes in Computer Science, Vol. 1219 (Springer, Berlin, 1997).

[18] J. Robinson, S.H. Russ, B. Flachs and B. Heckel, A task migration implementation for the message passing interface, in: *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Systems*, Syracuse, NY (August 1995).

[19] J.M. Smith, A survey of process migration mechanisms, Operating Systems Review 22(3) (July 1988) 28–40.

[20] P. Smith and N.C. Hutchinson, Heterogeneous process migration: The Tui system, Technical Report, University of British Columbia (February 28, 1996).

[21] Sun Microsystems, *External Data Representation Reference Manual* (Sun Microsystems, 1985).

[22] Sun Microsystems, Java Object Serialization Specification, Revision 0.9 (1996).

[23] B. Steensgaard and E. Jul, Object and native code thread mobility among heterogeneous computers, SOSP (1995).

[24] M.M. Theimer and B. Hayes, Heterogeneous process migration by recompilation, in: *Proceedings of the 11th International. Conference on Distributed Computing Systems*, Arlington, TX (May 1991) pp. 18–25.

[25] D.G. von Bank, C.M. Shub and R.W. Sebesta, A unified model of pointwise equivalence of procedural computations, ACM Transactions on Programming Languages and Systems 16(6) (November 1994) 1842–1874.

[26] H. Zhou and A. Geist, Receiver makes right data conversion in PVM, in: *Proceedings of 14th International Conference on Computers and Communications*, Phoenix (March 1995) pp. 458–464.

Adam Ferrari. Photograph and biography not available at time of publication.

Steve J. Chapin. Photograph and biography not available at time of publication.

Andrew Grimshaw. Photograph and biography not available at time of publication.