Process Introspection: A Heterogeneous Checkpoint/Restart Mechanism Based on Automatic Code Modification

Adam J. Ferrari
Stephen J. Chapin
Andrew S. Grimshaw

Technical Report CS-97-05
Department of Computer Science
University of Virginia
Charlottesville, VA 22903

March 25, 1997

Abstract

Process Introspection is a fundamentally new solution to the process checkpoint/restart problem suitable for use in high-performance heterogeneous distributed systems. A process checkpoint/restart mechanism for such an environment has the primary requirement that it must be platform-independent: process checkpoints produced on a computer system of one architecture or operating system platform must be restartable on a computer system of a different architecture or operating system platform. The central feature of the Process Introspection approach is automatic augmentation of program code to incorporate checkpoint and restart functionality. This program modification is performed at a platform-independent intermediate level of code representation, and preserves the original program semantics. This approach has attractive properties including portability, ease of use, customizability to application-specific requirements, and flexibility with respect to basic performance trade-offs. Our solution is novel in its true platform- and run-time-system-independence—no system support or non-portable code is required by our core mechanisms. Recent experimental results obtained using a prototype implementation of the Process Introspection system indicate the overheads introduced by the mechanisms are acceptable for computationally demanding applications.

1 Introduction

We have developed a fundamentally new approach to the problem of capturing the dynamic state of processes in a platform-independent manner called Process Introspection. Our approach centers on the semantics-preserving transformation of programs by a source code translator that automatically incorporates autonomous checkpoint and restart functionality into processes. This automatic program modification is performed entirely at a platform-independent intermediate level of code representation. We have found that Process Introspection, as described in this paper, has many attractive properties including portability, ease of use by the programmer, customizability to application-specific requirements, and flexibility with respect to basic performance trade-offs. Our solution is novel in its true platform and run-time-system independence—no system support or non-portable code is required by our core mechanisms that capture and restore process state.

In this paper we explain the design of a Process Introspection system (section 2), describe our complete prototype implementation (section 3), and discuss recent experimental results obtained using the implementation (section 4). These results indicate that in addition to the more general desirable attributes of the system design, Process Introspection can be employed with low overhead and achieve good performance. We then examine related work (section 5) and give concluding remarks (section 6). In the remainder of this section, we more thoroughly define the problem and give design goals for our solution.

1.1 Background

Recent developments in software systems and the growing availability of higher-performance computing and networking hardware have made commonplace the use of networks of workstations, personal computers, and supercomputers as virtual, distributed-memory parallel machines, or metasystems, for solving computationally demanding problems [5,6,7]. The combination of heterogeneous architectures and operating system platforms within a high-performance distributed metasystem gives rise to a number of problems not present in homogenous systems. The complexity of varying architectural features, such as data representation and instruction sets, and varying operating system features, such as process management and communication interfaces, must be masked from the application programmer. Heterogeneity also complicates existing problems in parallel and distributed systems, such as task placement, which may depend on processor speed and type, operating system revision, and network interconnection bandwidth and latency, among other factors. Despite the added complexity and challenges involved in heterogeneous distributed computing, the promise of increased performance afforded by a larger hardware base, along with the ability to increase performance by mapping sub-tasks of a computation to the most appropriate available hardware, called *superconcurrency* in Freund and Cornwell [4], make heterogeneous computing an active and promising area of research.

The presence of heterogeneous computing systems significantly complicates the design of a process checkpoint/restart 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 host. A substantial body of research demonstrates the utility and desirability of such a mechanism. Process or object migration policies supporting load sharing and/or fault tolerance can use a process state capture facility. Systems such as Time Warp [8] that rely on the ability to "roll back" a local computation to provide semantic guarantees, such as the causal ordering of message delivery, require a checkpoint/restart facility. A distributed system such as Legion [6] can use checkpoint/restart for resource management: if the number of active entities in a system becomes greater than can be efficiently supported, the system could temporarily preempt the execution of some processes by checkpointing and destroying the processes, then later restarting them[1]. Another possible application is 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.

While a number of distributed systems running on homogeneous processors exhibit some ability to capture and restore the state of a running process, this feature is conspicuously absent in most existing high-performance heterogeneous distributed systems. The additional complexity inherently introduced by heterogeneity is the main reason that few designs for such a facility have been developed to date. In the homogeneous case, checkpoint/restart mechanisms can simply and directly manipulate the state of a process without semantic analysis of that state. For example, the state of a Unix process is simply the contents of its address space, plus its process control block (register values, file descriptor table, etc.). These entities are already conveniently available to the Unix kernel, making the internal state of a Unix process trivial to checkpoint. As long as the process is restarted on the same kind of Unix system and processor on which the checkpoint was produced, the contents of the address space need not be interpreted by the kernel to restore the process. Unfortunately, the address space and kernel process control information would be meaningless if used to restart the process on a differing Unix implementation or architecture. Differences in data representation, instruction sets, address space sizes (e.g. 32-bit vs. 64-bit addressing), and address space structures might make the saved state completely unintelligible at restart time.

Clearly, the state of the process cannot be captured using the naive mechanism which suffices in the homogeneous case. For a checkpoint mechanism to operate in a heterogeneous environment, it must examine and capture the logical structure and meaning of the process address space contents, as well as any operating system specific information (e.g. open file tables or intra-process communication state). This prospect is somewhat daunting—the logical point in execution, call stack (or call stacks, if threads are used), complex data structures, values and logical structure of heap allocated memory, etc. all must be analyzed and checkpointed in a platform-independent format, masking data format differences, addressing differences, instruction set differences and so on.

1.2 Problem Definition

Many different approaches to the heterogeneous state capture/restore problem are possible. One straightforward approach is to use an interpreted language, as in the case of Java object serialization [14]. In such cases, the inter-

1. This is roughly equivalent to a uniprocessor system swapping out a process to decrease the page fault rate.

preter acts as a virtual machine which can artificially homogenize a system composed of heterogeneous elements. This solution is attractive for its simplicity, but fails to meet many applications' performance constraints. This is but one example of how our intended operating environment and our target application group constrain the space of acceptable solutions to the heterogeneous checkpoint/restart problem.

The environment of interest is a distributed system consisting of a variety of node computing systems. These nodes may be of differing processor types, architectures, and configurations, and may run operating systems of differing types, capabilities, and versions. Processes, as defined by the local node operating systems, run on the nodes, typically executing as native code due to performance considerations. Processes in the system cooperate using a network. The target application domain includes, but is not limited to, high-performance distributed memory parallelized scientific applications exhibiting medium to course granularity. We fully expect that our systems will experience node failures and exhibit inherent load imbalance due to resource sharing.

Our mechanisms will generate a checkpoint for an individual process running in the system, and then use that checkpoint to later restart an equivalent process from the produced checkpoint, perhaps on a different node from the one on which the original process was running when the checkpoint was created. This leads to a fundamental constraint: the mechanism should generate *platform-independent* checkpoints (i.e. checkpoints should be restartable on any available operating system or architecture platform of interest in the target distributed system).

In our solution design, we make certain assumptions about the program which comprises the process to be checkpointed. First, we assume that a set of source code *modules* is given that can be compiled and linked to construct a native-code representation of the program on all platforms in the target distributed system. We assume that some of these modules are known to be *platform-independent,* i.e. their correctness does not depend on any implementation details of any particular underlying architecture or operating system. These modules are expected to be provided as code in some high-level language. The target application area is high-performance scientific computing, thus the languages of interest include C, C++, and Fortran. In principle, however, any high level language which can be translated to executable instructions on the platforms of interest should be adequate. Other modules have either *inherent platform dependencies* (such as special operating system requirements—e.g. a file interface or network communication interface module) or *perceived platform dependencies* (such as programmer use of algorithmic optimizations based on available architectural features that may not be performed automatically by available optimizing compilers, e.g. blocking versus vector implementations). Process Introspection is based on modifying the given program modules in such a way that the resulting completely-linked program will be able to autonomously checkpoint and later restart on a different type of node.

1.3 Design Goals

We have the following design goals for our checkpoint/restart mechanism:

- Little or no programmer effort should be required to checkpoint and restart architecture-independent modules. A large subset of problems of interest for high-performance computing can be solved using platform-independent programs; consider for example the large number of scientific numerical applications that are coded in Fortran and are basically portable across standard Fortran implementations.
- A convenient user interface should be provided for rendering architecture-dependent modules checkpointable in a manner that is interoperable with automatically-checkpointable modules. This interface should give the programmer flexibility to customize and tune a module's checkpoint/restart mechanism when appropriate; at the same time, it should be easy to use.
- The mechanism should provide low checkpoint-request-service wait time—i.e. the time between a checkpoint request being delivered to a process and that process beginning to service the request should be significantly less than the time required to write the checkpoint. This precludes techniques such as waiting for the program to reach a known, simple, consistent state (e.g. waiting for a complex call stack to finish and return to the main function, checkpointing, then proceeding with the next iteration).
- The run-time overhead introduced by the mechanism should be low. In particular, if checkpoints are not performed during a certain period of execution, the checkpointable version of the code should not run significantly slower than an optimized, non-checkpointable version of the code over that period. This goal might be stated simply as, "Don't pay if you don't play."
- The checkpoint/restart mechanism should perform with comparable costs to a homogeneous environment checkpoint mechanism. For example, on a Unix system, the checkpoint of a running process should not take significantly longer than producing a core dump of the process. Similarly, the

checkpoint of a Unix process should be comparable in size to a core image of the process.

- The mechanism should be general in nature. That is, it should be appropriate for use with a wide variety of programs, written in a variety of languages, and solving a wide range of problems. This precludes special purpose toolkits such as those appropriate only for scientific problems of a certain nature (e.g. stencil algorithms). Furthermore, the mechanism should be usable in a variety of different heterogeneous environments (e.g., a PVM environment [5], a Condor-like system [10], etc.).

2 The Process Introspection Model

We now describe the Process Introspection approach to constructing a heterogeneous process checkpoint/restart mechanism. The key element of Process Introspection is the semantics-preserving modification of a program to incorporate checkpoint and restart functionality, giving processes the ability to autonomously write checkpoints and perform restarts. This is a fundamentally different approach than existing solutions to the process state capture problem, which typically are based on an external agent (e.g. the operating system) examining and checkpointing the state of a process. In this section we describe an abstract overview of the Process Introspection model and strategy. In Section 3 we describe how the model presented here has been realized in a working prototype implementation.

2.1 Overview

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 the Process Introspection transformations will be applied—likely by a compiler, but also possibly by a programmer. The first key modification to the intermediate program is the addition of poll points: points in the code at which the process determines if a checkpoint should be produced (analogous to Bus Stops in Heterogeneous Emerald [15]). At a poll point, a process can create a checkpoint if one has been requested. Certain parts of the process state are easily captured—for example, any global variable or heap allocated data structures must be written to the checkpoint, but these elements of the process state are globally addressable and are thus easy to manipulate. The key difficulty in creating the checkpoint is the capture of the state of the subroutine invocation stack. In the Process Introspection approach, the process utilizes the native "subroutine return" mechanism provided by the intermediate instruction set to checkpoint the stack. The current active subroutine 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 restarts, the process employs the native "subroutine call" mechanism provided by the intermediate instruction set. On a restart, the base subroutine restores its state, then calls the next subroutine in the checkpointed stack, which repeats this process until the stack is completely restored. Again, the global variables and heap allocated data structures must also be recovered from the checkpoint at restart time.

This method of performing checkpoints and restarts uses only common features of high-level procedural languages, and clearly could be employed directly by a programmer. The problem with coding such a strategy by hand is that it is a complex, error-prone task, and furthermore it is a task neither directly related to the actual problem the programmer is trying to solve nor within the area of expertise of our expected user community of domain scientists. Hand coding would thus likely lead to increased development and debugging time. Fortunately, for many programs, these modifications can be performed automatically by a source code translator.

2.2 Process Model

To more carefully define the Process Introspection mechanism, we must first develop a more formal definition of a process. In our system model, the fundamental meaning of a process is defined by its intermediate representation program, a platform-independent procedural representation of the process's code that may (or may not) be the result of front-end high-level language translation.

The intermediate representation of the program consists of a set of subprograms and a set of global variables. Each global variable is a named, globally-addressable, typed data block available throughout the lifetime of the program, and each subprogram consists of a sequence of instructions and a set of automatically-allocated variables available to an activation of the subprogram throughout its lifetime. We define six classes of instructions in the intermediate representation, which should be general enough to serve as a target for any high-level programming language. *State Modification* instructions alter the value of *state elements* (program variables) based on combinations of other state element values or constant values. *Branching* instructions alter the control flow of the program, perhaps

based on a logical combination of active state elements and constant values (i.e. conditional branch). *Subprogram Activation* instructions activate a given subprogram—i.e. the automatic variables for the subprogram are instantiated on a logical call stack, the location of the subprogram call is saved for use when the subprogram terminates, and control flow shifts to the first instruction in the subprogram (a conventional stack based execution model). *Subprogram Termination* instructions terminate the currently active subprogram and return to the saved location. *Dynamic Memory Management* instructions allocate and free memory in the form of dynamically-allocated state elements. *Input/Output* instructions read and write the values of state elements from or to the program's environment, respectively.

The meaning of the program, as defined by the intermediate representation, is as follows. When a program begins executing and becomes a process, it acquires an active state element set. State elements are statically-sized named memory blocks upon which the instructions in the program operate. The active state element set consists of the set of global variables for the program, the set of dynamically-allocated memory blocks in use by the process, and an activation stack of automatically-allocated variables and parameters associated with live subroutines. The process begins execution in an automatic base activation of a special start subroutine, and executes the instructions in this routine's sequence until the last instruction is executed or the activation explicitly terminates, at which point execution stops. The base subroutine activation in the course of execution may call any other available subroutines, which may in turn make calls, resulting in arbitrarily deep, perhaps recursive, activation stacks.

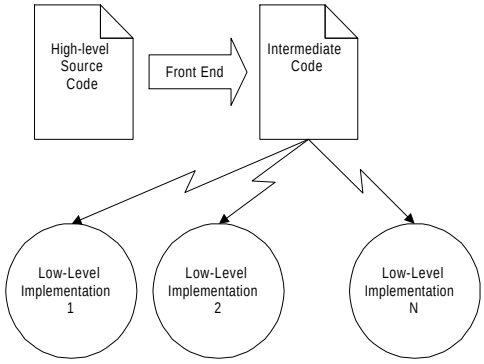

Figure 1. The program model

Given the intermediate-code representation that defines the meaning of the program, low-level program implementations are created. Any number of low-level implementations might be built, e.g. for different computing platforms, with different levels and types of debugging information or optimizations. It should be emphasized that low-level implementations are expected to be primarily native-code executables. These executables might be optimized to take advantage of certain properties which are not observable in the intermediate program—the intermediate representation defines the meaning of the program, but does not dictate the actual state elements or instruction stream used by any implementation of the program. As long as the implementation preserves the meaning of the program for all inputs (i.e., as long as the output is produced as specified by the intermediate representation), the implementation is correct. This leaves open the possibility of optimization for specific hardware features, exploitation of parallelism where possible, and so on.

2.3 Process Introspection

The Process Introspection checkpoint mechanism is based on the idea that the intermediate representation of the program can be modified to allow it to describe in complete detail certain intermediate states of its computation. In other words, the process is given the ability to periodically output the values of all active state elements (at the universal intermediate-representation level) along with a representation of the call stack that could be used to recover the logical location in control flow (i.e. to re-activate all active subprograms and return to the instruction at which the checkpoint was created). Our approach makes the following changes to the intermediate representation of the program:

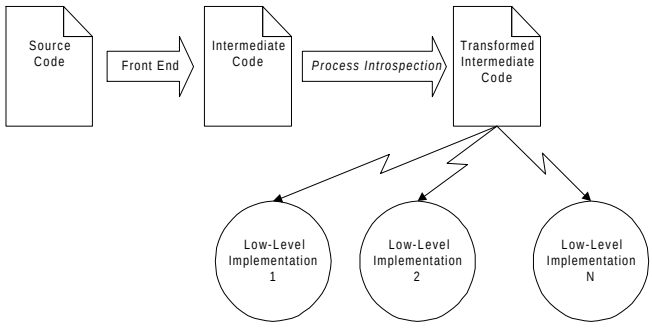

Figure 2. Creation of an introspective program

- A global variable is added called *CheckpointStatus*. This variable will be used by the program to indicate if a checkpoint has been requested (via an interrupt) or is in the process of being constructed. *CheckpointStatus* is initialized to indicate no checkpoint in progress; the handler code for the checkpoint request interrupt simply sets *CheckpointStatus* to indicate that a checkpoint request has been received.

- A new subroutine called P_{chkpt} is added to the program. P_{chkpt} contains instructions to output the values of all global variables and all active dynamically-allocated state elements.
- For P_{chkpt} to be able to output all of the dynamically-allocated state elements, a table must be maintained by the program reflecting the current complete set of active dynamically-allocated state elements. This requires that instructions be added after each dynamic allocation and deletion instruction to maintain this dynamic allocation table.
- After each subprogram activation instruction, an instruction is added to poll for a checkpoint request (i.e. examine the value of *CheckpointStatus*). If a checkpoint has been requested, instructions are executed to save the local subprogram activation. The program first writes all of the automatically-allocated state elements associated with the current subprogram to the checkpoint. Next, a marker indicating the logical location in control flow (i.e. the instruction location of the previous subprogram activation instruction) is output. If the current activation is the base subroutine for the program, a call to P_{chkpt} is also executed. Finally, a subprogram termination instruction is executed to cause an immediate return to the current subroutine's caller, which will execute similar instructions to save its activation, and so on down the stack. This inserted code is a *mandatory poll point*: a point at which the program must check for a checkpoint request. In fact, more frequent polls for checkpoint requests might be desirable. In this case, poll points as described above can be inserted between any two instructions in the original intermediate-code program. These poll points are called *optional poll points*.

These modifications to the program give it the ability to periodically poll for the receipt of a checkpoint request. If a checkpoint is requested, the program writes the values of its complete set of state elements along with the information necessary to reconstruct the process's control flow state (i.e. subroutine activation stack and location in the code).

For the checkpoint information constructed by the mechanism described above to be useful, the program must also be able to restart from a checkpoint. Our approach makes the following changes to the intermediate representation of the program:

- The *CheckpointStatus* variable also indicates whether a restart is in progress or not. It must be initialized by input from the program's environment at the beginning of the base subroutine for the program. In the event that a restart is requested at this point, the process can assume that its checkpoint is available for reading in a well-known location (e.g. in a given file or network connection).
- A new subprogram called $P_{restart}$ is added to the program. $P_{restart}$ contains instructions to read the values of all static state elements from the checkpoint and re-instantiate (allocate) the set of active dynamically-allocated state elements stored in the checkpoint.
- A prologue of statements is added to each subprogram that checks the value of *CheckpointStatus*. If *CheckpointStatus* is set to indicate a restart, the prologue reads the values of all automatically-allocated state elements associated with the subroutine from the checkpoint. If the subprogram is the initial activation of the base subprogram, an instruction to call $P_{restart}$ is also executed. Finally, the logical location in control flow in the current subprogram activation is read from the checkpoint, and the associated code location is the target of a branch instruction. This location may be a subprogram activation instruction, in which case the next subprogram in the stack is called and must restore its activation, and so on up the stack. The location may also be an optional poll point, in which case the restart is complete; *CheckpointStatus* can be cleared, and execution continues normally.

With these additions, the program can restore an intermediate state as produced by its own checkpoint mechanism. In particular, since the checkpoint and restart mechanisms are specified at the level of the universal intermediate representation, different implementations of the program can read and write one another's checkpoints, assuming the state elements and control flow information can be stored in a universally recognizable format, masking architectural issues such as data representation (cf. Sun XDR [13]).

Because we stated in Section 2.2 that low-level implementations of the program may be arbitrarily optimized (with the constraint that they preserve the meaning of the program), it might seem that the correctness of our state capture mechanism is suspect. For example, what if a poll point initiated a checkpoint in the middle of a loop that was ordered differently in different implementations? Might the state captured by one implementation be inconsistent with any legal state of some other implementation? In fact, we can show that all semantics-preserving implementations of the transformed intermediate code step through the same sequence of poll points in the same order, and would produce equivalent state images at corresponding points. Furthermore, we can show that given a captured state, any implementation can resume from the corresponding point in execution. These properties are consequences of the fact that the state capture and restore mechanisms are specified completely in terms of the intermediate representation,

and all low-level implementations must preserve the meaning of the intermediate code. In practice, the checkpoint/restart mechanism as specified in the intermediate code will result in dependencies at poll points that will cause optimizations across poll-points to be prohibited.

2.4 Optimizations

It should be noted that a side-effect of the described checkpoint mechanism 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 producing the checkpoint. We note an important optimization to the above mechanism: in the process of writing the state elements associated with each frame to the checkpoint, they should also be saved in the process's memory (e.g. in a specially-allocated dynamic state element). This permits the implementation of a quick stack recovery after the checkpoint is produced. Because the checkpoint mechanism is non-destructive to other state elements (i.e. globals and dynamically-allocated memory blocks), this optimization permits the process to proceed without unreasonable delay after a checkpoint.

A further optimization to the described code modification scheme is also possible. In the definition of mandatory poll points, we stated that poll points must be placed immediately following every subroutine call statement. In fact, if we have 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 very 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 a checkpoint had been requested while the function was executing, we can safely continue normal execution after the call returns before beginning to service the request.

2.5 External State

We have described a model through which the complete *internal state* of a process can be captured. In any realistic system, the full state of a process will consist not only of its internal state, but also of its relationships to external services. For example, a process may be communicating with any number of other processes that store its network address. At the time of its checkpoint, a process may be the destination of any number of messages that are in transit in the network system. The process may own the locks to certain resources such as databases or hardware devices. The set of possible *external state* of the process can be large and complex to capture in a checkpoint. Nonetheless, any real checkpoint/restart mechanism must take such external state into consideration if it is to have any real application. In our model, the capture of external state occurs in one of two general ways.

Case 1, System Support—In some cases, it is desirable or convenient for the process's environment (either operating system or metasystem) to provide some system support for checkpointing with respect to external state. For example, the MIST [2] system provides system support for checkpointing sets of processes communicating via the PVM interface.

Case 2, Wrapper Modules—In many environments, system support is neither available nor convenient to implement. In these cases it is desirable to embed the ability to checkpoint external state in the process itself. For example, a process may not have direct access to its file table, but if it used wrapper routines to access all file operations, the wrapper library could maintain an accurate internal representation of the process's file usage. Similarly, if wrapper routines were used for network communication, the process could use a protocol with its peers to determine external network state (for example, messages in transit at the checkpoint). Using wrapper routines to capture a process's external state is a technique that has been demonstrated as effective in projects such as Condor [10] and other such load-sharing tools used in homogeneous systems.

3 Implementation

3.1 Overview

We have constructed a prototype implementation of the Process Introspection system consisting of a Process Introspection Library (PIL), which provides the programmer and compiler an interface for writing checkpointable modules, and a source code translator called APrIL (Automatic application of the Process Introspection Library) which can automatically apply the Process Introspection transformations to architecture-independent modules written in ANSI C. The implementation has been tested on a variety of workstation and PC platforms, including Sun workstations running Solaris or SunOS 4.x, SGI workstations running IRIX 5.x, IBM RS/6000 workstations running AIX,

DEC Alpha workstations running OSF/1, and PC compatibles running Linux and Microsoft Windows 95/ NT.

Usage of the system generally takes one of two forms. In the case of modules which have architecture dependencies (for example, modules which are checkpointable wrappers around external services such as the file system), the modifications to support Process Introspection must be hand coded (as in the case of Module 1 in Figure 3). For such cases, the PIL interface provides basic services such as data-format-independent access to checkpoint data. The job of the programmer in these cases is to adhere to the model as described above (e.g. by periodically polling for checkpoint requests), and to provide a platform-independent mechanism for saving the data associated with the module. For example, in coding a checkpointable file interface module, the programmer would need to design a mechanism for recording the state of all open files in use by the process. If the module provided integer file descriptors to users, it might save a table indicating the associated file name and file pointer for each descriptor. On restart, the module would need to use the local file system interface to re-open the files associated with the descriptors and seek to the appropriate locations.

The expected use of the system is more automatic than this. In the case of platform-independent modules, the programmer writes the module code in ANSI C, and uses the interfaces of other checkpointable modules. For example, instead of using the Unix file system interface, the programmer might use the file interface mentioned above. The code for the architecture-independent module is then automatically rendered checkpointable by applying the APrIL source code translator, which also utilizes the PIL as a run-time interface. This usage mode, which corresponds to the case of Module 2 in Figure 3, requires no extra work on the part of the programmer to apply Process Introspection.

Given this general description of the two major system components and their intended usage, we can now more closely examine the implementation details of each.

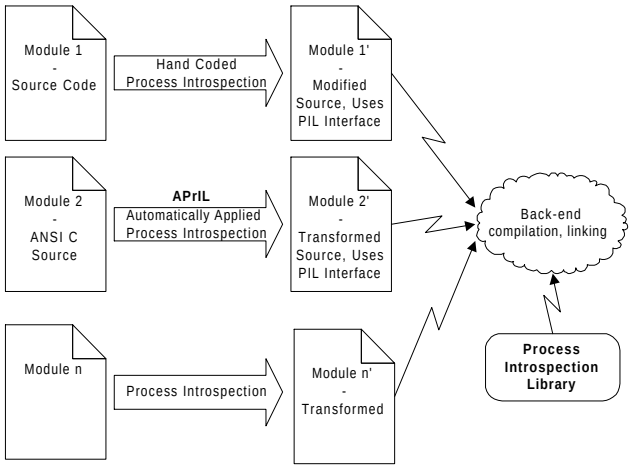

Figure 3. Using the Process Introspection system

3.2 The Process Introspection Library

The Process Introspection Library (PIL) is a central element of both usage modes for the system. In the case of hand-coded modules, the PIL provides the API for coding a module's checkpoint/restart capability. In the case of compiler-transformed modules, the PIL provides the 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 values. In addition, the library provides an event-based mechanism for coordinating the activities of modules at checkpoint and restart time. The key elements of the PIL are:

The Type Table—To checkpoint 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. Every memory block savable through the PIL interface should be describable as a linear vector of some number of elements of a type described by an entry in the type table. 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 run-time. The interface provides pre-defined type numbers for the basic types supported by ANSI C, and provides an interface for describing vectors and records based on existing types.

Data Format Conversion Module—The PIL provides an interface for reading and writing typed data from and to a checkpoint in an architecture-independent format, respectively. This interface is responsible for masking differences in byte ordering and floating point representation. When checkpointing memory blocks using the PIL interface, the library automatically includes in the checkpoint a description of the data formats used. Later, when the checkpoint is restored, the data format can be converted to the restarting processor's representation, a protocol known as *receiver-makes-right* [18]. 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 a single universal data format for checkpoints, and require conversion routines only between native formats and the universal format (reducing the complex-

ity to O(N) conversion routines for N formats)? In fact, the receiver-makes-right protocol makes sense only in light of the very small number of data formats actually used by current computer systems. By not requiring data format conversion on checkpoint, the cost of format conversions is avoided for the frequent case in which a checkpoint is restarted 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, when saved in a checkpoint they must be described using a logical format in place of the physical address. Similarly, at restore time, logical pointer descriptions saved in a checkpoint must be used to determine the physical memory address values that should be restored into all memory blocks. A suitable mechanism for this purpose can be based on the assignment of a unique identification number to every memory block of interest in the program. A logical pointer description then is a tuple containing a memory block identification tag, and an offset into the memory block. 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 implementation is based on simple case analysis; a pointer can be one of exactly five types:

- A reference into a heap allocated memory block
- A reference into a global memory block
- A reference into a local (stack) memory block
- A pointer to some code entry point
- A special value which has meaning in the program (such as NULL in C).

The PIL associates id numbers with memory blocks of each class (except the last), thus providing the basis for logical pointer description.

Global Variable Table—The PIL provides an automated mechanism for saving and restoring the values of global variables at checkpoint and restart time, respectively. The mechanism requires that the memory addresses, type table indices, and vector sizes of all globally-addressable memory blocks be registered with the PIL in a Global Variable Table. Besides providing automatic checkpoint and restart of globals, the Global Variable Table is used to perform pointer analysis as described above, handling the case where an address points to within a global memory block.

Heap Allocation Module—Similar to the case with globals, the PIL provides a mechanism for allocating memory blocks from the heap that will be automatically checkpointed and restored. The Heap Allocation module exports heap wrapper routines that perform typed memory block allocation and deallocation. These wrapper routines maintain a table of the addresses, type table indices, and vector sizes of all active dynamically-allocated memory blocks. In addition to supporting automatically checkpointed/restored dynamic memory blocks, this heap allocation table is used by the pointer analysis module.

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 vector 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 in the table, leaving open the possibility of register assignment.

Event Module—The Event Module provides the primary mechanism for modules to customize their checkpoint and restart behavior. The Event Module allows a program module to register function callbacks that will be automatically invoked by the system at checkpoint and restart time. To understand the importance of this module, consider the case of the file system module described previously. 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. On a restart, for example, it must use the local file interface to re-open the files that were in use at the checkpoint. It would also likely be responsible for maintaining the file version differences associated with each checkpoint. These extra activities would be coded in the form of event handlers which would be executed in response to checkpoint 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, with only the PIL in

hand, the programmer would be left to manually perform code modifications such as poll-point placement and prologue generation as described in Section 2.3. Fortunately, for platform-independent programs this process can be automated by a source code translator. Using the Sage++ toolkit [1], we have implemented this idea in the APrIL compiler. The model described in section Section 2.2 stated that the input to the translator be in a universal intermediate representation. In the case of APrIL, this intermediate code is simply ANSI C. APrIL takes as input ANSI C code, and produces as output new ANSI C code transformed as described in Section 2.3 and utilizing 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 specific transformations employed by APrIL.

3.3.1 Poll Points

In accordance with the model described in Section 2.3, APrIL inserts poll points throughout the code it transforms. At each poll point, code is inserted to check for an active checkpoint request. This simply involves examining the value of a global variable (**`PIL_ChkptStatus`**) which is set to indicate that a checkpoint request has been received (recall that this value is set an interrupt handler that fields external checkpoint requests). Immediately following the poll point, code is inserted which will be executed when a checkpoint is in progress. This code records the location in the frame at which the checkpoint is produced and jumps to a function epilogue that saves the actual parameters and locals 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 4.

```
_PIL_PollPt_1:
    if(PIL_ChkptStatus&PIL_ChkptNow) {
        PIL_PushCodeLocation(1);
        PIL_ChkptStatus|=PIL_ChkptInProgress;
        goto _PIL_save_frame_;
    }
```

Figure 4. An optional poll point

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 as described in Section 2.3. 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 checkpointing 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 the checkpoint was begun in a higher call frame), and one after the call site (in the event that the checkpoint is initiated immediately following a normal function return). An example of a mandatory poll point is given in Figure 5.

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

Figure 5. A mandatory poll point

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 request for a checkpoint sent to the process may suffer unseemly delay before being serviced. Clearly, a balanced approach based on the user's tolerance of introduced overhead and checkpoint-request wait time is required. If the user only expects to checkpoint infrequently (e.g. once every minute), but demands little introduced overhead, then very sparse, conservative poll-point placement is called for. Alternatively, if checkpoint-request wait

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 must extract functions from complex expressions and statements, and reduce them to simple C expression statements containing a single function call.

times must be very low (for example, if the checkpoint will be used for code migration to effect load sharing), more frequent, aggressive placement is appropriate. Unfortunately, the problem of statically examining code and determining the introduced overhead and resulting checkpoint 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. The implemented policies include:

- Mandatory poll points only
- Mandatory poll points plus optional poll points placed as the last statement in each nested loop (i.e. each loop that contains at least one other loop).
- Mandatory poll points plus optional poll points placed as the last statement in each outermost loop (i.e. a loop not contained in any other loop)
- Same as the previous, but poll points are added only to outermost loops with greater than *k* statements.
- Any combination of the above.

Many more policies are possible and are under consideration. 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 two of the currently available APrIL placement options (which are selected by specifying command line flags to the compiler) are examined in Section 4. These measurements can serve as an initial decision making tool for policy selection.

3.3.2 Function Prologues

As specified in the model described in Section 2.3, function prologues are added to every function definition transformed by APrIL. If any local variable or parameter addresses 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 a restart is in progress (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 checkpoint for this frame was created, and jump to the appropriate poll point label in the function. Figure 6 illustrates an APrIL function prologue transformation. The function heading given in Figure 6(a) is transformed to include the prologue depicted in Figure 6(b).

```
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_ChkptStatus&PIL_RestoreNow) {
      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 6. A function prologue transformation

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 this array is generated. The prologue checks the value of the **PIL_ChkptStatus** 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 checkpoint was performed is then jumped to using a **goto** based on a code location marker read from the checkpoint. For some code locations (those corresponding to optional poll points and the second label associated with mandatory poll points), the generated code first generates a call to the function **PIL_DoneRestart()** to complete the restart process and unset the **PIL_ChkptStatus** variable.

3.3.3 Function Epilogues

Poll points inserted by APrIL generate code to jump to a function epilogue in the event that a checkpoint is found to be in progress. The job of the function epilogue is 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 checkpoint 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 6. is depicted in Figure 7.

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

Figure 7. A function epilogue

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.

3.3.4 Module Initialization

The three types of transformations discussed thus far are primarily aimed at implementing the checkpoint and restoration of function call stacks. April also generates a routine to register any types defined by the module with the Type Table and register any globals defined by the 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.

3.3.5 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 **`malloc()`** C library routine, uses the parameter to **`malloc()`** to determine the allocation size, and determines the allocation type first on the type that the return value of **`malloc()`** is cast to (if it is available), and (failing that), second on the type of the variable to which the **`malloc()`** return value is assigned. This process will certainly fail if the memory allocation method used does not match our expected pattern. Future work will include investigating better heuristics for finding and wrapping heap allocations.

4 Performance Experiments

To examine the performance characteristics of our prototype implementation, we applied the system to a set of numerical applications. This set of test programs included:

- MM (Matrix Multiply)—Computes the product of two dense, square matrices of 512x512 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 which is applied to a 100x100 grid of solution elements until the change in the two-norm of the solution is less than 10^{-2}.
- GE (Gaussian Elimination)—Performs Gaussian elimination with partial pivoting on a dense 256x256 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.
- QS (Quicksort)—Applies a quicksort algorithm to an array of 2^{21} integers.

We used three very different computer systems as test platforms for our experiments. The first is a 150 Mhz Intel Pentium-based system with 32 MB of RAM, running Linux 2.0 and using the GNU gcc version 2.7.2 compiler. The second is an IBM RS/6000-250 PowerPC-based system with 128 MB of RAM, running AIX 4.2 and the associated xlc 3.1 compiler. The third system is a Sun SparcStation-20/514 with 512 MB of RAM running SunOS 5.5.1 and the SPARCCompiler C 3.0 compiler (Sun cc).

Our first experiment was performed to measure the run-time overhead introduced by our code transformations. These transformations can add overhead not only because of extra instructions executed, but can also affect the ability

of optimizing compilers to apply certain transformations. The impact of our code transformations on 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 the available set of heuristic poll-point-placement policies supported by APrIL to each of out test applications. We report results for two out of the six available transformation policies, as the other policies produced results quite similar to one of those reported. The selected transformation heuristics are:

- Conservative—This is the more conservative of these two placement policies, placing mandatory poll points and optional poll points only in nested loops (i.e. loops that contain other loops).
- Aggressive—This more aggressive policy places mandatory poll points, poll points in nested loops, and poll points in all outermost loops (i.e. loops that are not contained in another loop).

We took each of the test programs transformed using both of these policies, as well as the original non-transformed (and thus, non-checkpointable) programs, and compiled each with and without optimization (optimization was specified using the "-O" flag in all cases). We ran each of the resulting executables to completion without performing checkpoints or restarts during execution, timing the complete run time including time to load the process. We expect that the more aggressive poll-point-placement policy will lead to increased overhead and possibly reduced effectiveness of optimizations (i.e. less realized speedup compared to the non-optimized code). The results of these tests are presented in Table 1.

Table 1: Performance overhead, times in seconds

		MM	GS	GE	CG	QS
i586, Linux, gcc	***No transforms***	99.8	70.0	24.2	45.2	12.0
	No trans opt	72.6	28.4	10.2	30.6	6.0
	Conservative	99.9	71.0	24.3	45.4	12.4
	Conserv opt	77.8	29.0	10.5	31.6	6.8
	Aggressive	99.9	75.4	24.3	48.1	12.4
	Aggressive opt	77.8	35.2	10.5	38.5	6.8
PowerPC, AIX, xlc	***No transforms***	266.5	112.3	30.8	77.9	22.9
	No trans opt	201.9	53.4	9.1	36.9	11.6
	Conservative	266.6	117.0	30.8	83.7	26.5
	Conserv opt	202.7	54.0	9.2	42.9	14.6
	Aggressive	274.6	127.2	30.8	96.6	26.7
	Aggressive opt	214.8	54.2	9.2	44.8	14.8
Sparc-10, Solaris, cc	***No transforms***	316.8	175.2	54.0	172.8	33.5
	No trans opt	229.3	59.0	18.5	128.3	16.8
	Conservative	318.5	176.8	55.3	176.5	35.3
	Conserv opt	234.7	59.5	20.7	128.3	17.1
	Aggressive	318.3	204.2	55.4	179.2	45.1
	Aggressive opt	234.0	70.9	20.0	136.1	18.6

The first trend that we observe in these results is that the overhead of our transformations under a conservative placement policy is generally low (around 10% on average), both in terms of additional work incurred and impedance of optimization. We note that the impact if the poll-point-placement policy is dependent on the application. This statement will generally hold for relatively simple heuristic policies such as those described here. Fortunately, these simple heuristics appear able to deliver an acceptable level of overhead. A further trend we observe in Table 1 is that while the more aggressive placement policy incurs higher overhead in most cases, this effect is also application-dependent. A more aggressive policy will often lead to higher overhead, but can be used to reduce checkpoint request wait time.

We performed a second set of experiments to determine the amount of time that a checkpoint request would

have to wait before receiving service. Recall, in the model, a checkpoint request is sent to the process, interrupts the process, and sets a global variable indicating that a checkpoint has been requested. It is only later, when the process reaches a poll point, that the checkpoint actually begins. This naturally leads to the question, if a process is sent a checkpoint request, how long will it be before a poll point is reached and the checkpoint writing begins? To investigate the checkpoint request wait time resulting from our system, we modified the APrIL compiler to add performance instrumentation to transformed code at each poll-point. At each poll point, the time is marked, the time since the last poll point is measured and added to a running sum, and the count of the number of poll points is incremented. On process exit, the average time between poll points is recorded. We compiled the transformed and instrumented versions of each application for both poll-point-placement policies with optimization on, expecting that more aggressive poll-point placement would lead to lower average poll point intervals and checkpoint wait times. Results are presented in Table 2. Note that the described mechanism for measuring the intervals between poll points can only accurately measure intervals longer than a system-specific threshold. Times in Table 2 in parentheses should be considered upper bounds below which our timing mechanism could not measure.

Table 2: Average interval between poll points, times in milliseconds

		MM	GS	GE	CG	QS
i586, Linux, gcc	***Conservative***	0.298	0.038	0.041	0.357	(0.004)
	Aggressive	0.151	(0.004)	0.040	(0.004)	(0.004)
PowerPC, AIX, xlc	***Conservative***	0.783	0.060	0.037	0.474	(0.003)
	Aggressive	0.393	(0.003)	0.036	(0.003)	(0.003)
Sparc-10, Solaris, cc	***Conservative***	0.749	0.065	0.070	1.284	(0.004)
	Aggressive	0.371	(0.004)	0.061	(0.004)	(0.004)

The general trend we note in this set of experiments is that, although a process in our system must poll for checkpoint requests, the average interval between polls is very small (typically < 1 millisecond), especially when compared with the cost that will generally be associated with actual checkpoint creation. As expected, the more aggressive placement policy leads to a reduction in the average time interval between poll points. This, paired with our overhead measurements presented in Table 1, leads us to the non-startling observation that run-time overhead can be traded off for lower checkpoint-request wait times in our system. We also note that relatively naive use of the system generally results in attractive performance characteristics—i.e. we need not add a great deal of overhead to the process to achieve an average checkpoint-request wait time that is orders of magnitude less than the cost of checkpoint transmission, either over a network or to a disk.

Table 3: Checkpoint/Restart Costs, times in milliseconds

		MM	GS	GE	CG	QS
Checkpoint Size		6291682	160207	535032	2097495	8388802
i586, Linux, gcc	***Checkpoint***	362.5	10.8	63.7	128.3	514.1
	Restart	615.3	16.5	85.5	198.0	834.8
PowerPC, AIX, xlc	***Checkpoint***	1498.2	128.8	258.6	565.3	2029.8
	Restart	779.9	24.8	149.3	259.9	1032.0
Sparc-10, Solaris, cc	***Checkpoint***	945.5	22.1	96.5	259.2	1396.5
	Restart	1091.9	30.8	154.7	351.3	1002.6

Our final set of experiments was performed to examine the efficiency of the checkpoint and restart mechanisms. For these tests, we instrumented the PIL to note the time when either a checkpoint or restart is initiated, and to subsequently record the time of completion. To obtain repeatable results, we also instrumented the PIL to automatically force a checkpoint request after a set number of poll points are encountered. We ran each of our transformed test applications (compiled with optimization) until 1000 poll points were encountered. At that point, a checkpoint was

written to disk, and the process was terminated. We then used the produced checkpoint to time a restart from disk using a newly created process. Although tests were run using all of the available poll-point-placement policies, the results did not vary measurably for the different cases. The results presented in Table 3 reflect the use of the "Conservative" placement strategy from the previous experiments.

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, Shub, and Sebesta in [17]. In this paper, the authors identify the general idea that a procedural computation (essentially as described in Section 2.2) 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 [3]. 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 [16]. 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 [15]. 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 checkpointable programs, we do not require support from any external agent for checkpoint/restart 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 checkpointable wrapper interface for system services that maintain external state for processes.

A similar approach to that of heterogeneous Emerald called Tui [12] 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 state of a running process.

6 Conclusions

We have presented Process Introspection, a novel design and implementation of a heterogeneous process checkpoint/restart mechanism based on automatic code modification. Our initial round of experiments using the prototype implementation of the system has produced encouraging results. First, we find that relatively naive application of a simple poll-point-placement policy can achieve acceptable levels of incurred overhead while at the same time providing good performance in terms of average checkpoint-request wait time. Furthermore, results indicate that by varying

the frequency of poll-point placement (i.e. by throttling the aggressiveness of the placement policy), we can allow the user to trade performance overhead for checkpoint wait time performance. This offers the user the flexibility to meet demanding performance requirements either in terms of limiting overhead or providing quick checkpoint responsiveness.

Beyond this performance oriented flexibility, our system provides additional attractive features. The core internal state capture mechanism described is highly portable, requiring no special run-time system support. Support of an additional platform type requires no modification to APrIL, and at most addition of support for an additional data format in the PIL. Furthermore, our mechanism is general—we believe it could be utilized in a variety of different distributed system environments. For example, we are currently working on adapting the system for use in the Legion [9] wide-area, object-oriented distributed system, and are also investigating integration into a PVM [5] or MPI [7] 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.

References

[1] 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.

[2] J. Casas, D. L. Clark, P. S. Galbiati, R. Konuru, S. W. Otto, R. M. Prouty, J. Walpole, "MIST: PVM with Transparent Migration and Checkpointing", 3rd Annual PVM Users' Group Meeting, Pittsburgh, PA, May 7-9, 1995.

[3] F.B. Dubach, R.M. Rutherford, and C.M. Shub, "Process-Originated Migration in a Heterogeneous Environment", Proceedings of the ACM Computer Science Conference, pp.98-102, Feb. 1989.

[4] R.F. Freund and D. S. Cornwell, "Superconcurrency: A Form of Distributed Heterogeneous Supercomputing," *Supercomputing Review*, Vol. 3, pp. 47-50, Oct. 1990.

[5] A. Geist, A Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam, PVM: Parallel Virtual Machine, MIT Press, 1994.

[6] 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*, pp. 257-270, Vol. 21, No. 3, June 1994.

[7] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, 1994.

[8] D.R. Jefferson, "Virtual Time", *ACM Transaction on Programming Languages and Systems*, Vol. 7, No. 3, pp.404-425, July 1985.

[9] M.J. Lewis, A.S. Grimshaw, "The Core Legion Object Model," *Proceedings of IEEE High Performance Distributed Computing 5*, pp. 551-561 Syracuse, NY, August 6-9, 1996.

[10] 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*, pp. 104-111, 1988.

[11] J. Robinson, S.H. Russ, B. Flachs, and B. Heckel, "A Task Migration Implementation for the Message Passing Interface", *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Systems*, Syracuse, NY, August, 1995.

[12] P. Smith and N.C. Hutchinson, "Heterogeneous Process Migration: The Tui System", Technical Report, University of British Columbia, Feb. 28, 1996.

[13] Sun Microsystems, *External Data Representation Reference Manual*, Sun Microsystems, Jan. 1985.

[14] Sun Microsystems, *Java Object Serialization Specification*, Revision 0.9, 1996.

[15] B. Steensgaard and E. Jul, "Object and Native Code Thread Mobility Among Heterogeneous Computers", SOSP 1995.

[16] M.M. Theimer, and B. Hayes, "Heterogeneous Process Migration by Recompilation," *Proceedings of the 11th International. Conference on Distributed Computing Systems*, Arlington, TX, pp. 18-25, May 1991.

[17] 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*, Vol. 16, No. 6, pp. 1842-1874, Nov. 1994.

[18] H. Zhou and A. Geist "Receiver Makes Right Data Conversion in PVM", *Proceedings of 14th International Conference on Computers and Communications*, Phoenix, pp. 458-464, March 1995.