Process Introspection: A Checkpoint Mechanism for High Performance Heterogeneous Distributed Systems

Adam J. Ferrari

Technical Report CS-96-15
Department of Computer Science
University of Virginia
Charlottesville, VA 22903
ferrari@virginia.edu

October 10, 1996

Abstract

The Process Introspection project is a design and implementation effort, the main goal of which is to construct a general purpose, flexible, efficient checkpoint/restart mechanism appropriate for use in high performance heterogeneous distributed systems. This checkpoint/restart mechanism has the primary constraint that it must be platform independent; that is, checkpoints produced on one architecture or operating system platform must be restartable on a different architecture or operating system platform. The Process Introspection mechanism is based on a design pattern for constructing interoperable checkpointable modules. Application of the design pattern is automated by two levels of software tools: a library of support routines that facilitate the use of the design pattern, and a source code translator that automatically applies the pattern to platform independent modules. A prototype implementation of library has been constructed and used to demonstrate that the design pattern can be applied effectively to construct platform independent checkpointable programs that operate efficiently.

1 Introduction

This document details the design, prototype implementation, and initial evaluation of a flexible, efficient mechanism for checkpointing and restarting processes in a heterogeneous distributed environment. A checkpoint/restart mechanism is a significant element of a distributed system as it is required to implement a number of basic fault tolerance and load balancing schemes. The current trend towards heterogeneity in high performance distributed systems has given rise to the necessity that the checkpoint/restart mechanism employed be platform independent. This requirement greatly increases the complexity of the checkpoint/restart problem, since it requires that the state of a running process be capturable in a form that can be reinstantiated on a completely different architecture and operating system platform with a different instruction set, data format, address space layout, etc. The mechanism to perform platform independent checkpoint/restart described here is based on the notion of *process introspection*, the fundamental fact that any program can be modified to examine and capture its own internal dynamic state in a form that can later be

recovered by an equivalent program running on a different architecture. In particular, the design discussed utilizes support in language translation systems and runtime libraries to produce executable programs with the ability to capture and restore their own internal state in a platform independent form. In many cases, this scheme requires little or no work on the part of the programmer.

Current systems or designs for platform independent checkpoint/restart and migration mechanisms suffer a number of significant limitations, including serious performance degradation, lack of portability, and lack of support for user optimization or specification of special module checkpoint mechanisms. The mechanism described here is based on a flexible abstract design pattern for constructing checkpointable programs, the application of which is supported by a set of software tools to automate much of the work that would otherwise be performed by the programmer. The design described here constitutes the first flexible, portable, extensible, truly platform independent checkpoint/restart mechanism appropriate for high performance heterogeneous distributed environments. It addresses the major problems and limitations of existing approaches, and when fully implemented will provide a much needed but missing element of functionality in high performance heterogeneous distributed systems.

1.1 Background

Recent developments in software systems and the growing availability of higher performance computing and networking hardware have made the use of networks of workstations, personal computers, and supercomputers as virtual distributed memory parallel machines commonplace for solving computationally demanding problems[15,17]. Furthermore, developments in distributed systems technology have made routine the use of heterogeneous collections of computing systems by a single application. These trends have combined to produce the emerging field of high performance, heterogeneous distributed computing. The combination of heterogeneous architecture and operating system platforms to produce a usable high performance distributed meta-system gives rise to a number of problems not present in homogenous systems. For example, 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 user. Furthermore, heterogeneity complicates existing problems in parallel and distributed systems. For example, the ability to create a good schedule for a parallel computation is complicated by the presence of processors and interconnection networks of varying speeds and capabilities. 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 concept of "superconcurrency,"[13] the ability to increase performance by mapping sub-tasks of a computation to the most appropriate available hardware, make heterogeneous computing an active and promising area of research.

A problem that is significantly complicated by the presence of heterogeneous computing systems is the design of a checkpoint/restart mechanism, a facility to 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. This feature is conspicuously absent in existing heterogeneous distributed systems although it exists in some form in a number of distributed systems running on homogeneous sets of processors. The uses of a checkpoint/restart mechanism have been the subject of a great deal of research. For example, a number of process or object migration policies to support load balancing, fault tolerance, or both require an automatic checkpointing facility. Other fault tolerance schemes such as distributed checkpoint/restart methods are further examples. An automatic checkpointing scheme can be used to implement the semantics of certain programming environments such as Time Warp[18] and other optimistic processing systems that rely on the ability to "roll back" a local computation to provide causal guarantees about the order of message delivery at a process. Some systems require the ability to checkpoint active entities to support scheduling and load balancing activities. For example, if the number of active entities in a system becomes greater than can be efficiently supported, the system might benefit from the ability to temporarily preempt the execution of some processes by checkpointing and destroying them, then later restarting them from checkpoints. Beyond these existing uses, many other uses for an automatic platform independent checkpoint

mechanism can be imagined. For example, the ability to spontaneously replicate a running process might be useful for performance enhancement or for augmenting existing replication based fault tolerance schemes[3]. Another possibility is platform independent debugging through techniques such as using checkpoints to replay a process from a given point in execution or statically examining the state of a process as captured in a checkpoint.

An automatic checkpoint/restart mechanism would be useful in heterogeneous systems for the same reasons it has been found an invaluable tool in homogeneous systems. The basic reason that no adequate design for such a facility has been developed to date is the relative difficulty due to the inherent additional complexity introduced by heterogeneity. In the homogeneous case, the state of a process can be manipulated simply and directly without being analyzed by the checkpoint/restart mechanism. For example, the state of a Unix process is simply the contents of its address space, plus its process control block (register values, file table, intra-process communication buffers, 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 on the same kind of processor on which the checkpoint was produced, the contents of the address space need not be interpreted by the kernel in order to restore the process. Unfortunately, the address space and kernel process control information would be meaningless if used to restart the process on a different Unix implementation running on a different architecture. The data representation might be different, thereby making any data in the address space meaningless. The code segment of the address space would be unintelligible given the different instruction set. The address space may in fact be of a different size (e.g. 32 vs. 64 bit addressing), and may be laid out in an entirely incompatible manner (e.g. text, data, bss locations, segment and page sizes, etc.).

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, intra-process communication buffers and information). This prospect is somewhat daunting - the logical point in execution and the call stacks of all threads of control, complex data structures including those linked by pointers, the 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. In fact, this problem is not automatically solvable in the fully general case; a program can always be constructed to have inherent architecture or operating system dependencies. This does not mean that the situation is hopeless, but we must restrict the problem somewhat to arrive at a useful solution from an engineering perspective. Many problems can be solved using programs that are efficiently checkpointable in an architecture independent format. The issues of interest revolve around how automatic, general, flexible, and efficient the mechanism can be made in order for it to be useful for checkpointing real programs running in real systems with performance constraints and limited programmer ability and time.

2 Problem Statement

2.1 Context

Many different approaches to the heterogeneous checkpoint problem are possible. For example, one straightforward approach is to use an interpreted language, as in [5]. This reduces the problem of checkpointing arbitrary programs in an architecture independent format to the problem of checkpointing a single program - the interpreter. This solution is attractive for its simplicity, but is bound to fail to meet many applications' performance constraints. To constrain the possible solution space to the problem that we will state in this section, we must first describe the context in which the solution is to operate.

The environment of interest is a distributed system consisting of a variety of node computing systems. These nodes may be of different processor types, architectures, and configurations, and may run operating systems of different types, capabilities, and versions. Processes as defined by the local node operating sys-

tems run on the nodes, typically executing as native code if this is permitted by the node operating systems in order to support high performance, computation intensive applications. Processes in the system will typically 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. Distributed systems of primary interest will exhibit node failures as well as inherent load imbalance due to resource sharing.

2.2 Problem Definition

We assume that the following are given:

- A set of *modules* that can be combined to construct a native code representation of a processing entity on all platforms of interest in some target distributed system.
- Some modules are known to be *platform independent*, i.e. their correctness does not depend on any implementation details of a given 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).

We wish to design a mechanism to compose the given modules into a complete program that can be automatically checkpointed and later restarted from the checkpoint, perhaps on a different type of node from the one on which the checkpoint was produced. The most fundamental constraint on the mechanism is that it 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).

2.3 Solution Quality Metrics

Any checkpoint/restart mechanism that is intended for use in the environment described in Section 2.1 and that solves the problem described in Section 2.2 can be evaluated by examining the degree to which it satisfies the following quality metrics:

- Ideally, little or no programmer effort should be required in order to checkpoint and restart architecture independent modules. Consequently, no programmer effort should be required to automatically checkpoint and restart programs consisting of only architecture independent modules. It should be noted that most problems of interest for high performance computing can be solved using platform independent programs; consider for example the large number of numerical kernels that are written in Fortran and that are basically portable across standard Fortran implementations.
- A convenient programmer interface should be provided for modifying architecture dependent modules in order to render them checkpointable in a manner that is interoperable with compiler modified modules. The programming interface should give the programmer a high degree of flexibility to customize and tune a module's checkpoint/restart mechanism; at the same time, it should afford enough power to make it easy to use.
- The mechanism should provide low checkpoint request service latency - 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

some “iteration number,” and then proceeding with the next complex “iteration”).

- The run-time overhead introduced by the mechanism should be low. In particular, if checkpoints are not performed during execution, the checkpointable version of the code should not run significantly slower than an optimized, non-checkpointable version of the code. This metric might be stated simply as, “Don’t pay if you don’t play.”
- The checkpoint/restart mechanism should perform with a comparable cost 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.
- The checkpoints produced by the mechanism should not be unreasonably large. Continuing the above example, the checkpoint of a Unix process should not be significantly larger than 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 codes).
- The mechanism should be demonstrably integrable (without heroic effort) into existing heterogeneous distributed systems that are used to run “real” production quality applications.

3 Design Overview

We wish to design a software system solving the problem described in Section 2, and to analyze the degree to which the system meets each of the design goals enumerated in Section 2.3. The design solution described in this paper is based on the idea of *process introspection* - the ability of a process to examine and describe its own internal state in a logical, platform independent format. In some senses, all processes that employ a custom programmed checkpoint/restart implementation utilize the concept of process introspection. The system described here extends this technique of hand coding checkpoint/restart functionality for individual processes into an integrated approach in which the development of introspective, checkpointable program modules is completely automated when possible, or is at least rendered significantly less complex through the use of library tools and a general design pattern when a handed coded checkpoint facility for a module is still appropriate. The system design consists of the following components:

- **The Process Introspection Design Pattern**, a design template for writing checkpointable codes. This design pattern describes the elements that must be added to a program in order for it to support introspective checkpointing, as well as the relationships and responsibilities of these elements.
- Dynamic runtime support via the **Process Introspection Library** (**PIL**), a set of tools to automate or simplify many of the tasks involved in implementing an architecture independent introspective checkpoint mechanism for a module.
- The **Automatic Process Introspection Compiler** (**APrIL**), a source code translator that can transform architecture independent modules specified in a high level language into introspective checkpointable modules that utilize the PIL for interoperability.
- A **Standard Checkpoint Interface** (**SCI**) specification mechanism for describing the checkpoint/restart interfaces to modules which will be linked to produce an introspective process.
- A **Central Checkpoint Coordinator** (**CCC**) module interface definition. The CCC is the part of an introspective process that coordinates the operation of the introspective modules in the process at checkpoint/restart time. The CCC also provides the public checkpoint interface for the introspective process; that is, the interface via which the process can be asked to produce a checkpoint or to restart from a given checkpoint.

In addition to these core elements of the proposed system design, a number of incidental elements will eventually be designed and produced. In particular, a number of pre-provided checkpointable modules must be constructed in order to allow introspective processes to be conveniently integrated into existing

distributed systems (e.g. a checkpointable message passing module). Furthermore, checkpointable modules implementing such typically needed tools as file access will be provided to enhance the usability of the system.

This paper describes the design, implementation, and analysis issues involved in constructing a system consisting of the components described above. In Section 4, we discuss the Process Introspection Design pattern. In Section 5, we examine the interface and implementation issues related to the Process Introspection Library runtime support system, a set of tools that facilitate the application of the Process Introspection Design Pattern. In Section 6, we discuss the automated application of process introspection through source compilation techniques via the APrIL compiler. In Section 7, we describe the Central Checkpoint Coordinator and the Standard Checkpoint Interface. Section 8 describes initial implementation and application experiences with the Process Introspection Library and design pattern, and includes a discussion of preliminary performance results obtained with a prototype version of the system. Section 9 discusses related systems and theoretical results, and Section 10 offers concluding remarks and describes current and future research directions.

4 The Process Introspection Design Pattern

The most concrete elements of the system design described in this paper are software tools that aid in the implementation of platform independent checkpointable programs. However, the fundamental theoretical basis for these tools is a design pattern for constructing checkpointable programs appropriate for use in the target environment. A Design Pattern consists of four elements[14]:

- A **name**, in this case, the Process Introspection Design Pattern.
- A **target problem**, in this case, the problem described in Section 2.
- A **solution**, consisting of elements that make up the design and the ways that they interact.
- A set of **consequences**, the trade-offs and implications of using the pattern.

In this section, we will describe the general solution and consequences associated with the Process Introspection Design Pattern.

4.1 Process Model

Before describing the Process Introspection Design Pattern (or the software tools designed to help apply it), we must first define the process model assumed. Processes as viewed by the system are defined as follows:

- A process executes a program. Equivalent executable versions of the process's program are available for all platforms of interest. This is a somewhat flexible constraint - not *all* architecture specific binaries need to be available as long as the scheduling/migration policies can be constrained to utilize feasible schedules given the available set of binaries.
- The executable program associated with a process is built from a set of modules. Each module contains a set of executable subprograms and data.
- A running process contains a set of active threads of control. The execution model for these threads is based on the traditional procedural model with a stack based parameter and local variable storage scheme. Thus, a thread of control in the system logically consists of a program counter (i.e. a logical location in the executable code for the process) and a subprogram activation stack containing the local variable and parameter storage for the thread's subprogram call stack.
- A process contains data in the form of memory blocks. Every memory block contains some structured layout of elements of basic data types (i.e. data types supported by the processor and/or programming language systems used by the process). A process contains three basic classes of memory blocks: statically sized global blocks, dynamically allocated blocks whose size and structure are determined at run time, and automatically allocated stack blocks that are

local to some subprogram activation in a thread of control within the process.

4.2 The Process Introspection Design Pattern

Given the definition of process discussed in Section 4.1, and the problem of checkpointing or restarting the process as stated in Section 2, we can now describe a general design pattern for modifying the modules that make up the process in order to support introspective checkpoint and restart. These general design and implementation strategies can be applied by a programmer by hand to create the checkpoint/restart mechanism for the process. Given the code for the process (i.e for the modules that make up its program), the following two principles are applied:

1. The ability to save and restore the call stacks of all threads must be added to the program.

To implement the checkpoint feature for threads, the code that the threads execute should be modified to periodically poll for checkpoint requests; i.e. **poll points** should be placed throughout the code. As we noted in Section 2.3, low checkpoint request service latency is a quality metric of the checkpoint mechanism, so poll points ought to be placed in the code frequently enough to meet the performance needs of the application. On the other hand, poll points constitute additional run time overhead, so excessive placement of poll points would be un-wise.

If a checkpoint request is encountered at a poll point, the thread must immediately checkpoint any data in its active call frame along with its logical point in execution, and returns to its calling subroutine which must then save its call frame and return, and so on. In this way, the threads each checkpoint their stacks using the native subroutine return mechanism to traverse the call frames.

Similarly, to implement the restart feature, the native subroutine call mechanism is employed. When the process is asked to restart from a given checkpoint, it must call the initial subroutine for all threads active at the checkpoint. Each initial subroutine restores its local variables from the checkpoint, uses control flow to advance to the correct logical point in execution (as reflected in the checkpoint), and then calls the next subroutine in the checkpointed stack. The called subroutine then repeats the process, which continues until the final active subroutine is called and can proceed from its checkpointed logical point in execution.

2. The ability to save and restore all memory blocks must be added to the program.

The above mechanism requires that a means to save and restore the memory blocks in the stack be added to the program. More generally, a mechanism must be added to the program to save all of the memory blocks in the process - global, dynamically allocated, as well as the stacks. The key attribute of this mechanism is that it must be capable of masking all platform dependencies in all kinds of program data. For example, the checkpointed data should not depend on any particular low level data format (e.g. “little endian” versus “big endian” integers). More generally, however, the checkpointed data must be made to mask higher level dissimilarities in the format and interpretation of memory blocks on different platforms. For example, a file interface on one operating system might represent an open file as an integer file descriptor. This integer would be meaningless to the file interface module on some other platform. Thus, the data (in this case, an open file description) would have to be checkpointed using a higher level description. In this example, the higher level description might include a file name and current location in the file. Another example in which higher level descriptions are needed is in the checkpointing of pointers. A memory address on one platform would need to be modified to reflect differences in address space layout and data format on another platform. Thus, if pointers were to occur in a memory block, they would need to be checkpointed using a logical description of the address they referred to instead of a low level memory location.

The primary consequence of the above modifications to a program implementing a process as described in Section 4.1 is that the process will be capable of introspective checkpoints and restarts. Fur-

thermore, the checkpoints produced by such a program will be platform independent. As mentioned, the placement of poll points is critical to the application of the pattern; too few can result in high checkpoint request service latency, while too many can result in high run-time overhead. Both of these risks must be illuminated as potentially negative, but avoidable, consequences of the pattern.

It is evident even from this general description that making the described modifications to a program manually could be very complex, even for relatively simple programs containing a small number of modules and few platform dependencies. Increasingly complex codes would be correspondingly more difficult to modify in order to support introspection. The difficulty involved in applying of the Process Introspection Design Pattern could be considered a negative consequence. This leads to the need to provide software tools to automate as many of the tasks involved in making a program capable of introspective checkpoint/restart as possible.

5 Process Introspection Library and Interface

The Process Introspection Library (PIL) is the most basic software layer intended to automate aspects of the Process Introspection Design Pattern for checkpointable programs. The PIL presents an API that can be used to deploy the coding strategies described in Section 4.2, which allow programs to be automatically checkpointable. The API is intended to be used by programmers to facilitate the hand coding of checkpoint algorithms, but it must also provide efficient access to runtime services suitable for use by a compiler which can provide a higher level of automation. We now examine the design issues involved in the main PIL modules.

5.1 Logical Program Counter Stack

The set of subroutine invocation stacks associated with a process define a logical point of execution (called a Logical Program Counter (LPC) value) in each active call frame. Each thread of control in the process has a stack of LPC values which defines its logical point in execution (i.e. its set of active subprograms and the logical point in execution within each one). The LPC stack of each thread must be computed at checkpoint time and stored with the checkpoint to restore the thread at restart time, using the native subroutine scheme described in Section 4.2. Similarly, at restore time, when the native subroutine call mechanism is used to reinstantiate the call stacks, the logical program counter value for each stack frame must be made available so that each stack frame will be able to determine the correct logical point at which to continue execution. The Logical Program Counter Stack module provides an interface for accomplishing these tasks. Note, the LPC Stack module does *not* restore the call stacks automatically; the threads of control owning the call stacks must restore the physical stacks themselves, but can use the LPC Stack module to determine where in each frame to continue execution, thus simplifying the task.

5.2 Thread Management Module

To ensure that the appropriate threads of control can be restarted from the appropriate entry points, a checkpoint must record a description of the threads that were active at the time the checkpoint was produced. Each thread of control in a process must be registered with the Thread Management Module which is responsible for checkpointing information about the active threads of control. The Thread Management module is also responsible for exporting a platform independent threads interface so that the semantics of threads-related operations (such as synchronization and scheduling) can be made consistent across different operating systems.

5.3 Data Format Conversion Module

When checkpointing memory blocks using the PIL, a process includes a description of the data formats used. Later, when the checkpoint is restored, the data format can be converted to the restarting pro-

cessor's representation, a protocol known as "receiver makes right"[36]. The PIL supports a set of routines to perform data format conversion on buffered data using this "receiver makes right" strategy. Consequently, this library module must contain routines to translate a set of supported data types from every available format to every other available format. This $O(N^2)$ (where N is the number of different data formats) requirement initially seems like a bad idea; 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 complexity 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 used by current and planned computer systems. By not requiring data format conversion on checkpoint, the cost of format conversions is avoided for the common case in which a checkpoint is restarted on a similar type of machine to the one on which it was created.

5.4 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 mapping type identifier numbers 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 standard compiler, except that it is available and dynamically configurable at runtime.

5.5 Pointer Analysis Module

Memory addresses (i.e. pointers) contained within memory blocks must be described using a logical format in the checkpoint. Similarly, at restore time, logical pointer descriptions must be used to determine the physical memory address values that should be restored into all memory blocks. A suitable mechanism for this purpose is based on the assignment of a unique identification number to every heap, global, and stack memory block. A logical pointer description then is a tuple containing a memory block identification tag, and an offset into the memory block. Offsets specified in a checkpoint may need to be adjusted due to data format differences on the restarting processor; again, this is a receiver makes right protocol. It should be noted that instead of pointing into a heap, stack, or global memory block, a memory address might point to some code location or might contain an invalid address although being typed to contain a pointer (e.g. a nil value). These kinds of values must also be in the logical pointer description space.

The Pointer Analysis Module provides the mechanisms for describing the logical value of memory locations (pointers) in a checkpoint, and for interpreting these logical values into actual memory addresses at restart time. These mechanisms are 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 Pointer Analysis module examines physical addresses to determine which of these kinds the pointer is of (assuming the final case if the first four fail). It produces logical descriptions of pointers each of which contains a logical identification number (that can be used to determine its type and base location at restart time) and an offset based on the checkpointer's data format.

5.6 Global Variable Table

The memory addresses, type table indices, and vector sizes of all globally addressable memory blocks must be registered with the PIL. The Global Variable Table provides an interface to accomplish this. It also

exports routines that can be used by the system to save the values of all global memory blocks to a checkpoint buffer, and to restore those values using the type description table and data format conversion routines.

5.7 Heap Allocation Table

In addition to globals, the addresses, type table indices, and vector sizes of all active dynamically allocated memory blocks must be registered with the PIL. To accomplish this, the Heap Allocation module exports wrapper routines that should be used for typed memory block allocation and de-allocation. These routines maintain a table of all active heap allocated memory blocks. As with the Global Variable Table, the Heap Allocation Table must export routines to save and restore the state of all heap allocated blocks.

5.8 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 are assigned a logical identification number via the Code Location Table interface. During the program's execution, code location pointers can be described logically in terms of code location table indices.

5.9 Active Local Variable Table

Since pointers can refer to local variables, the addresses, type table indices, and vector sizes of *some* local variable memory blocks must be registered with the PIL. Note, only those locals whose addresses can ever be assigned to pointers (and whose address can consequently be found in some memory block) need to be registered with the Active Local Variable Table. This leaves open the possibility that local variables can be stored in registers.

A prototype of the PIL supporting most of the functionality described in this section has been implemented and is operational. It demonstrates that the discussed design pattern can be applied without great difficulty to implement checkpoint/restart mechanisms for actual programs. Implementation details and preliminary performance results obtained using this prototype PIL are described in Section 8.

6 Automatic Process Introspection Compilation Techniques

The PIL makes hand coding introspective checkpoint/restart mechanisms for modules significantly less complex, but in many cases a higher level of automation is possible. If a module is specified in a high level language (e.g. C or Fortran) and contains no platform dependencies, PIL calls can be automatically inserted into transformed code, completely automating the coding of the checkpoint/restart mechanism in the best case, and requiring very little programmer input at a high level in the worst case. It should be immediately noted that this automated usage of the PIL will not be appropriate for some modules. For example, the file interface module already mentioned will likely not be specifiable in a platform independent manner since calls to a lower level operating system will likely be needed. In cases such as this, the programmer is left to design and implement the checkpoint/restart mechanism for the module using the PIL. Fortunately, many modules such as the file interface example can be designed and coded once, and then can be frequently reused by automatically translated, platform independent modules.

In this section, we provide an overview of the design of the "Automated usage of the Process Introspection Library" (APrIL) Compiler, a source code translator to automatically apply the Process Introspection Design Pattern. In particular, we will examine the general goals of the APrIL translator, and examine the most fundamental APrIL code transformations in detail. Implementation of the APrIL compiler is the subject of current ongoing work.

6.1 Translation Goals

The “Automated usage of the Process Introspection Library” (APrIL) compiler will provide an automatic, programmer transparent mechanism for employing the PIL library to render a module checkpointable. Use of APrIL is predicated on the programmer providing a module implementation in a high level language (e.g. C or Fortran) in an architecture independent form. The APrIL compiler translates this architecture independent high level language source code into a lower level but architecture independent intermediate representation. This intermediate representation is transformed to utilize the PIL runtime system in a manner consistent with the Process Introspection Design pattern described in Section 4.2. The transformed intermediate representation is then used to generate object code (which may be optimized) for all platforms of interest. The APrIL code transformations on the intermediate representation are based on three central goals:

1. The module must keep the PIL run-time tables consistent with the actual state of the process during execution. This is accomplished by inserting code to update the tables at points in the code where state changes must be reflected in the tables. The API presented by the PIL is designed to avoid frequent updates to the run-time tables. As much of the work as possible is concentrated at process startup, checkpoint, and restart times, in keeping with the “don’t pay if you don’t play” philosophy. Examples of runtime table maintenance routines that must be inserted by APrIL include:
    - On process start-up, the type table, global variable table, and function entry point table must be initialized.
    - When a subroutine begins execution, it must register in the Active Local Variable Table the memory addresses and type table indices of any local variables whose addresses can be assigned to pointers.
2. According to the design pattern described in Section 4.2, threads executing code in the module must periodically poll for checkpoint requests during execution. This requires that APrIL place poll points throughout the module code. In case the poll indicates that a checkpoint must be generated, the introspective checkpoint mechanism must be dispatched (i.e. the memory blocks in the current call frame must be saved and an immediate subroutine return must be performed). An open research issue is where in the code and how frequently poll points should be inserted. The more poll points that are placed in the code, the lower the latency between checkpoint request and service will be. On the other hand, more poll points will lead to higher runtime overhead and more constraints on the back-end optimizer.
3. When a restart is requested, the process must restore all threads of execution. Using the design pattern described in Section 4.2, this requires subroutines to determine if a restart is in progress when it begins executing, and to restore the state of the stack frame if a restart is in fact occurring.

6.2 Intermediate Representation

Before developing specific code transformations that will be used to accomplish the goals described above, the issue of which intermediate representation on which APrIL will operate must be addressed. A large variety of intermediate code representations have been proposed and are in common use in production and research compilers. For example, abstract syntax trees, program dependence graphs, and assembly language-like virtual machine instructions (e.g. byte-codes) are common possibilities. The choice of an intermediate representation for APrIL would preferably balance the following goals:

- Architecture independence
- High-level language independence
- Good existing tool support (e.g. front-ends for various languages, back-ends for various platforms, optimizers, etc.)

The current APrIL design choice in this area is to use source-to-source translation, making a subset of ANSI C serve as the intermediate language. ANSI C meets the first requirement, architecture independence, if certain platform dependent features (e.g. the “asm” directive) are excluded from the allowable

subset. While seemingly counter-intuitive, ANSI C also meets the second requirement of high-level language independence. For example, source-to-source tools exist to translate C++, Fortran, and Pascal (among others) to ANSI C. Finally, source-to-source translation based on ANSI C meets the third goal of a rich existing tool set. Back-end technology is available via existing high quality optimizing compilers. Front-end source-to-source translation tool-kits are also available; for example the Sage++ library[6] offers an object-oriented interface to parsing, manipulating, and transforming C using a set of C++ object classes.

It should be noted that the assumption of a high-level language as the intermediate representation is not fundamental to the APrIL design. If a different intermediate representation were used, equivalent transformations to those which will be described in the remainder of this section could be used to implement the automatic checkpoint code.

6.3 APrIL Transformations

We will now examine the most fundamental specific transformations employed by APrIL. The primary goals of these transformation are those discussed in Section 6.1.

6.3.1 Function Prologues

Function prologues are added to every function definition transformed by APrIL. If any local variable addresses are assigned to pointers 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 stack restart is in progress (recall, stack restarts using the PIL are implemented using the normal function call mechanism). APrIL generates code to be executed in case of a restart which will restore values of local variables and actual parameters, determine the Logical Program Counter location at which the checkpoint for this frame was created, and jump to a label in the function body corresponding to the LPC. The following example illustrates an APrIL function prologue. The function beginning:

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

is transformed to include the prologue:

```
void example(double *A)
{
        int i;
        double temp[100];
        PIL_RegisterStackPointer(temp,PIL_Double,100);
        if(PIL_CheckpointStatus&PIL_StatusRestoreNow) {
                int PIL_rst_pnt = PIL_PopLPCValue();
                A = PIL_RestoreStackPointer();
                i = PIL_RestoreStackInt();
                PIL_RestoreStackDoubles(temp,100);
                switch(PIL_rst_pnt) {
                        case 1: PIL_DoneRestart(); goto _PIL_PollPt_1;
                        case 2: goto _PIL_PollPt_2;
                        case 3: PIL_DoneRestart(); goto _PIL_PollPt_3;
                }
        }
```

This function has an array which is evidently later assigned to a pointer, hence the call to register the address, size, and type of the “temp” array. The prologue then checks the value of the special variable “PIL_CheckpointStatus” to determine if this function call was made in the process of restoring a call stack. If it was, the LPC value, actual parameter, and locals are restored using PIL routines. The correct point in the function is then jumped to using a “goto” based on the LPC value.

6.3.2 Poll Points

APrIL must insert poll points throughout the code it transforms. At each poll point, code must be generated to poll to see if a checkpoint is in progress. Code must be generated to be executed if a checkpoint is in progress which will record the logical program counter value for the frame and jump to a function epilogue which will save the actual parameters and locals. APrIL generates two kinds of poll points: standard and mandatory function call site poll points. Standard poll points can be inserted in the transformed code between any two statements. A standard poll point has a single labeled LPC value and performs the poll described above. An example of a standard poll point that might be found in the example function above is:

```
_PIL_PollPt_1:
       if(PIL_CheckpointStatus&PIL_StatusCheckpointNow) {
              PIL_PushLPCValue(1);
              PIL_CheckpointStatus|=PIL_StatusCheckpointInProgress;
              goto _PIL_save_frame_;
       }
```

Mandatory function call point poll points are required to be inserted by APrIL after every function call statement in the code[1]. Mandatory function call poll points are required in order to implement the stack save mechanism based on the native function return mechanism. Every function return in APrIL transformed code can be an actual return or a return that is used to save the stack. The mandatory function call point must catch and implement the latter case. This requires two LPC values for a mandatory poll point, one before the call site (in the case that the checkpoint began in a higher call frame), and one after the call site (in the event that the checkpoint should begin immediately following a normal function return). For example, a mandatory poll point that might occur in the transformed example code is:

```
_PIL_PollPt_2:
       i = function(A,temp,100);
_PIL_PollPt_3:
       if(PIL_CheckpointStatus&PIL_StatusCheckpointNow) {
              if(PIL_CheckpointStatus&PIL_StatusCheckpointInProgress)
                PIL_PushLPCValue(2);
              else {
                PIL_PushLPCValue(3);
                PIL_CheckpointStatus|=PIL_StatusCheckpointInProgress;
              }
              goto _PIL_save_frame_;
       }
```

6.3.3 Function Epilogues

The poll points inserted by APrIL generate code to jump to a function epilogue in the event that a checkpoint is found to be progress. Thus, APrIL generates an epilogue for each function it transforms placed beyond the last return statement (the epilogue is accessible only by goto, and is not normally executed by a standard function return). If the epilogue is executed (i.e. jumped to from a poll point), it saves the stack frame variables and actual parameters, and returns to the next function activation leaving the “Checkpoint in progress” flag set so that the stack save will continue. The function epilogue for our example function would be:

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

1. Of course, function calls can occur in expressions, in which case they must be extracted from the expression and assigned to a temporary variable.

6.3.4 Module Initialization

The three types of transformations discussed thus far are primarily aimed at implementing the checkpoint and restoration of function call stacks. A routine to register any types defined by the module with the Type Table and insert any globals defined by the module in the Global Variable Tables is also generated by APrIL. The generation of this function is a straightforward process based on any types, globals, and static variables found in the module. The mechanism by which this initialization function is marked as a module initializer to be called by the PIL on process startup is discussed in Section 7, which describes the Standard Checkpoint Interface.

6.3.5 Heap Function Transformations

One of the more difficult transformations that APrIL must perform 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 definition, APrIL will need to perform a heuristic to determine when heap allocation is taking place, and the type and size of the allocated memory. Of course, such a process will not be accurate or even possible in all cases, and thus user input (or at least verification of results produced using a heuristic) may be required at certain points during translation. For example, one possible heuristic could find all calls to the `malloc()` C library routine, use the parameter to `malloc()` to determine the allocation size, and base the allocation type determination on the type of value the result is casted or assigned to.

7 Checkpoint Coordination and Module Interfaces

Assuming modules are automatically transformed or are hand coded to support introspective checkpoints using the PIL interface and general design pattern, they must still be made to work together to produce checkpoints. In particular, it will certainly be desirable to combine separately developed and compiled modules to construct complete applications; these separately developed and compiled modules must be linkable and interoperable. The interoperation of modules to introspectively produce checkpoints or restart processes is achieved by a combination of a Central Checkpoint Coordinator module and a Standard Checkpoint Interface for introspective modules.

The Standard Checkpoint Interface (SCI) is a functional interface that must be exported by all of the modules that are linked together to construct an introspective process. The SCI essentially defines a set of (possibly empty) function callbacks that are executed when certain key events relevant for checkpoint/restart purposes occur. The list of SCI events includes:

- A *Process Startup* event is generated every time the process starts, either for the first time or at restart time. The startup event handler for each module is responsible for registering any globals and/or data type definitions included in the module.
- A *Checkpoint Start* event is generated when a checkpoint has been requested. During the handling of this event, a consistent description of the module suitable for writing to the checkpoint should be created. The module should not discard any information that other modules might depend on at this point. For example, if a heap allocated block owned by the module might be addressed by some other module, it should not be freed at during the Checkpoint Start event handler.
- A *Checkpoint End* event is generated after the Checkpoint Start event has been handled (i.e. the checkpoint has been constructed). During the handling of this event, any resources held by the module can be freed.
- A *Restart Start* event is generated when a restart has been requested. At this point, the module should attempt to restore its state from the checkpoint. A complete restoration may not be possible if needed information has not yet been restored by other modules. When this is the case, the module should record the need to later restore some of its state during the Restart Done event handling.
- A *Restart Done* event is generated after all Restart Start event handlers have completed. During the handling of this event, modules should restore any state dependent on information not available during

the Restart Start event handling.

The SCI will be implemented as a separate “meta-file” that must accompany a compiled module object or library file naming the event handlers contained in the module. Linkage of SCI modules will be performed by a special linker tool that will generate code to register all handlers for the appropriate events.

The SCI is utilized by a Central Checkpoint coordinator module. This module implements the public interface to the program’s checkpoint/restart facility. It must export a mechanism by which checkpoints or restarts can be requested, and must coordinate the service of these requests. Service coordination by the CCC is essentially the act of calling the registered event handlers of all modules whenever their associated events occur.

8 Preliminary Experiments and Results

Prototype implementations of the PIL and a simple CCC module were recently constructed and used as the basis for feasibility demonstrations and initial performance and cost analysis. In particular, three sample applications were hand transformed using the proposed APrIL source-to-source compilation techniques to examine typical runtime overheads, to gain an initial insight into the impact on back end optimizations, and to determine checkpoint request service latencies (i.e. the time between checkpoint request and service). Two numerical kernels (matrix multiplication and a sparse Gauss-Seidel solver) were selected as typical target codes, and a quicksort kernel was selected as a potentially high run-time overhead example.

As an initial demonstration of the feasibility of the process introspection technique, each of the example programs was compiled, run, and verified as checkpointable/restartable across Sun workstations running Solaris or SunOS 4.x, SGI workstations running IRIX 5.x, IBM RS/6000 workstations running AIX, DEC Alpha workstations running OSF1, and PC compatibles running Linux, Microsoft Windows NT, and Microsoft Windows 95. The interface selected for the simple CCC overloads the “control-C” interrupt of a process to checkpoint and exit the running program instead of simply terminating it. The CCC writes the checkpoint to the process’s current working directory using a well known file name. Later, when the program is run again, the CCC notes the presence of a checkpoint, and uses it to implement a restart instead of allowing the process to start up normally.

In addition to these feasibility demonstrations, the programs were compiled with and without optimizations, and run on a range of problem sizes. Each sample run was allowed to complete without performing any checkpoints or restarts in order to measure the introduced run-time overhead and affects on the back-end optimizer. Non-checkpointable versions of the codes were also run as control cases. Since all of the measured performance metrics of the sample programs depend on the placement of poll points, a simple placement heuristic was selected; poll points were placed at basic block boundaries, except in the innermost loop of multiply nested loops. Of course, many different placement strategies are possible and could produce different performance trade-offs. This simple heuristic was selected because it would be very simple to implement in the APrIL compiler (and thus will be at least one of the supported placement policies for APrIL), and it leads to a reasonable balance between introduced run-time overheads and checkpoint request latencies. In all test cases, the average checkpoint service latencies were measured and found to range from 0.01 to 1.0 milliseconds, indicating the utilized poll point placement scheme results in checkpoint latencies at least an order of magnitude below the time to save a checkpoint.

Representative performance results of these are presented below. In each table, “Normal” indicates the execution time of the non-optimized, nontransformed (i.e. not checkpointable) code. “Transformed” indicates the execution time of the transformed (checkpointable) code compiled with no optimizations. “Optimized” indicates the execution time of the optimized, non-transformed code, while “Trans. Opt.” indicates the execution time of the transformed code compiled with optimization. All times are listed in seconds, and represent the best run time taken over 8 runs to reduce the affect of shared resource contention.

Table 1: NxN Matrix multiplication, Ultrasparc, compiled with gcc

N	32	64	128	256	512
Normal	0.03	0.16	2.49	27.36	115.77
Transformed	0.11	0.33	2.92	27.76	121.42
Optimized	0.06	0.08	0.81	19.86	75.72
Trans. Opt.	0.10	0.14	1.00	19.88	76.41

Table 2: NxN Matrix multiplication, RS/6000, compiled with xlc

N	32	64	128	256	512
Normal	0.03	0.26	2.66	21.16	288.96
Transformed	0.03	0.27	2.66	21.17	288.99
Optimized	0.01	0.06	1.16	9.14	198.01
Trans. Opt.	0.01	0.07	1.16	9.18	199.95

Table 3: 2D Gauss-Seidel Solver, NxN grid, Ultrasparc, compiled with gcc

N	32	64	128	256
Normal	0.12	2.29	44.53	854.93
Transformed	0.12	2.29	46.78	859.35
Optimized	0.11	0.37	16.26	239.60
Trans. Opt.	0.11	0.39	16.72	267.50

Table 4: 2D Gauss-Seidel Solver, NxN grid, SGI, compiled with cc

N	32	64	128	256
Normal	0.22	2.27	40.87	670.96
Transformed	0.22	2.29	41.54	685.57
Optimized	0.12	0.86	17.40	268.75
Trans. Opt.	0.20	0.95	19.12	313.44

Table 5: Quicksort, 2^N keys, Ultrasparc, compiled with gcc

N	17	18	19	20	21
Normal	1.95	3.28	6.86	13.91	28.25
Transformed	1.96	3.68	7.54	15.31	31.25
Optimized	0.83	1.20	2.48	4.92	9.85
Trans. Opt.	1.00	1.46	2.99	5.94	12.22

Table 6: Quicksort, 2^N keys, SGI, compiled with cc

N	17	18	19	20	21
Normal	1.87	3.81	8.22	17.22	35.44
Transformed	2.20	4.55	9.73	20.41	42.15
Optimized	0.70	1.42	3.05	6.42	13.16
Trans. Opt.	1.04	2.12	4.26	8.96	18.50

The results of the performance tests demonstrate three general cases that can occur for a given placement policy. The first test, matrix multiplication, exhibited little difference in the performance of the checkpointable and non-checkpointable code, both for non-optimized and optimized versions. This indicates that the utilized poll placement heuristic is a good choice for the matrix multiplication code, as it results in low overhead and low checkpoint request latency, and also does not affect the operation of the optimizer. The next case, gauss-seidel, illustrates a case where the introduced overhead is low, but the optimizer is somewhat constrained. While the non-optimized checkpointable and non-checkpointable codes execute in roughly the same times, the optimized checkpointable code runs 10%-15% slower than the optimized non-checkpointable code. This indicates that the extra work introduced in this example is minimal, as it is in the matrix multiply example, but the optimizer is less able to speed up the code due to the placement of poll points. The final example, quicksort, illustrates the case where poll points are too frequent (10's of microseconds apart), and overhead is introduced due to the extra work. In this case, the effect of the added overhead rather than any constraints on the optimizer result again in a 10%-15% slowdown of the application. These three cases illustrate the critical nature of the poll point placement algorithm. If the poll points are well placed, the code will not suffer performance degradation and will exhibit low checkpoint service latency. On the other hand, poor placement or too liberal placement of poll points can seriously affect performance.

9 Related Work

The problem of checkpointing a process in a platform independent manner is closely related to work in two key areas: process migration, and object/data structure marshalling and migration. We will now examine related work in these areas.

9.1 Related Work in Homogeneous Process Migration

The area of process migration has been the subject of a great deal of research, both in terms of mechanisms and policies. The work most closely related to this proposal deals with migration mechanisms,

which in general can be viewed as checkpoint mechanisms. A broad survey of migration mechanisms in homogeneous environments is presented in [27]. Most homogeneous migration mechanisms are implemented at the kernel level of distributed and network operating systems. Examples of such systems include Charlotte[1], Sprite[10], DEMOS/MP[25], and the V-System[33]. In all of these migration mechanisms, the address space (i.e. the internal state) of the process is assumed migratable without modification, and the focus of the work is generally on masking differences in OS services (e.g. file system, intra-process communication) from the migrated process. Besides the obvious difference of the assumption of a homogeneous environment, these systems differ from the introspective mechanism in that they require kernel support. The argument for implementing process migration or checkpoint at the kernel level is based on the desire for efficiency and the availability of process resource usage information at the kernel level. In the context of a large scale, heterogeneous distributed system, these arguments become weaker. First, as the number of different architecture and operating system platforms grows, the issue of mechanism portability becomes important in addition to efficiency concerns. Furthermore, in meta-system approaches[16], the assumption that the operating system will have the needed intra-process communication information associated with a process becomes false, and the design decision to modify the kernels of all included hosts becomes impractical. These facts lead to the desire for a user-level checkpoint/migration mechanism.

User level process migration schemes have been implemented in some existing distributed systems[8,21,22,26]. For example, the Condor[21] load balancing system migrates processes in homogeneous environments by transferring a core image of the process. Needed operating system specific information associated with the process is maintained at the user level by tracking the parameters and return values of all system calls. While this approach is typically somewhat less efficient than kernel level implementations, the Condor system is easily portable to any Unix-based platform. Common constraints on user level migration schemes in network operating systems such as Condor and [22] are that the underlying operating systems provide network transparent file access, and that migrated processes do not use intra-process communication mechanisms. Some systems such as MIST[8] and Fail-safe PVM[19] have overcome many of these limitations. The primary difference between existing user-level approaches and the introspective mechanism is the lack of support for heterogeneity. In existing user level schemes (as in most existing kernel-level schemes), the address space image of the process is assumed to be migratable without translation. Again, the focus is on masking operating system service difference from migrated processes, rather than on dealing with the internal structure of the process. Furthermore, the limitations placed on operating system service usage in some of these systems are artifacts of the implementations, and do not reflect inherent limitations on user-level migration schemes. The introspective mechanism offers the flexibility to implement inherently operating system dependent interfaces such as message passing so they may be checkpointable.

9.2 Related Work in Heterogeneous Process Migration

The theoretical basis for process migration in a heterogeneous environment is discussed in [35]. In this work, the authors develop a formal definition of the points during execution at which a procedural computation based on some high level specification can be transformed to continue execution on any other Turing equivalent machine. They also introduce the idea that a compiler could place such points in a translated program, a key idea utilized by APrIL, which inserts poll points at which the PIL can correctly checkpoint the running process. Furthermore, the authors identify the fundamental trade-off between consistency point frequency and checkpoint request service latency discussed in Section 4.2. Thus, much of the theoretical foundation that demonstrates the feasibility of, and issues related to, a heterogeneous checkpoint mechanism has begun to be examined. The key element not yet addressed is the design and implementation of a flexible, reasonably easy to use system for real heterogeneous computing environments.

There have been a number of notable previous attempts at designing a heterogeneous process checkpoint or migration scheme. The first, an extension of the V migration mechanism, is presented in [11]. In this scheme, compiler support is used to generate meta-information about a process describing the locations and types of data items to be modified at migration time to mask data representation differences. For

example, a map of the data area and type information about heap allocated blocks is maintained as in the PIL mechanism. There are a number of key differences between the introspective checkpoint mechanism and the heterogeneous V mechanism. First, the V mechanism requires kernel support for migration. This has the drawbacks in a large scale heterogeneous system already described. Second, the V mechanism requires that data be stored at the same address in all migrated versions of the process - a constraint that may not be efficient or even possible to meet in some heterogeneous environments. The source of these limitations of the V approach is its basis in a traditional process migration scheme, and its focus on simply data-format converting and transferring the process address space to perform migration. The PIL focus on logical program structure enables it to operate at the user level, and without limitations on the actual memory image differences on different architectures.

Another approach to heterogeneous migration is presented by Theimer and Hayes in [34]. The basic idea of their scheme is to construct an intermediate source code representation of a running process at the point of migration, and to recompile this source at the migration target, continuing its execution in a semantics preserving manner. The migration source code produced in this scheme is required to contain code to reconstruct the migrated process to its state at the point of migration, as well as the normal code to implement the future operation of the process. The scheme proposed to generate this migration code is based on utilizing the debugger interface to examine all available process state information. The primary differences between this approach and the introspective approach are portability and efficiency. To implement the Theimer-Hayes approach, the low-level, non-portable debugger interface must be utilized by an process-external agent on each supported platform. Furthermore, the Theimer-Hayes approach requires the use of compilation at run time, which can result in significant additional migration latency. This approach was never implemented, and so actual performance and usability levels are difficult to evaluate.

A more recent approach to the heterogeneous process migration problem is the Tui system presented in [28]. This system also utilizes compiler support, extending the scheme of utilizing the debugging interface to examine or restore the state of a running process. While this system addresses some of the limitations of the two previous proposals, and has been more fully implemented in a prototype system, it has some drawbacks compared to the introspective approach. First, as with other schemes that modify backend compilers or utilize the debugging interface, the portability of the system can be limited, a major drawback in a truly heterogeneous system. Furthermore, the Tui approach requires all checkpointable code to be translated by a special compiler, unlike the introspective approach, which supports a programmer accessible API. The Tui approach thus limits or precludes the checkpointability of inherently architecture or operating system dependent modules, and may preclude programmer optimization of the checkpoint mechanism for some modules.

9.3 Work in Object Persistence and Migration

An additional source of related work deals with object and data structure marshalling and migration. Perhaps the most basic, low level data marshalling interface is the External Data Representation (XDR) library utilized by the Sun RPC mechanism[29]. This library provides routines to marshall atomic data types (e.g. integers, floating point values, etc.) into a machine independent format.

A higher level object marshalling and migration environment is provided by the Distributed Object Migration Environment system (Dome)[2]. This library is based on the C++ template mechanism, which it uses to support automatically migratable objects and vectors. This system hides the low-level programming details required to marshall and migrate distributed data structures, but it requires programmer declaration of checkpointable data structures, and does not support arbitrary checkpointable data types (e.g. a vector of floats is supported, but a linked list of records is not). Furthermore, this system constrains the programmer to a single language (C++), limits the kinds of data structures that can be checkpointed, and constrains the programming model to SPMD data-parallel codes.

A general framework for persistent objects was developed for the Arjuna[7,24] persistent object store. All persistent object classes in the system are required to define the mandatory member functions “save

state" and "restore state." These methods create and interpret objects of the class ObjectState which implements an object checkpoint/persistent representation. A limitation of this framework is that objects can only be checkpointed at the well defined point of method invocation. Furthermore, the generation of the checkpoint code is left to the programmer.

A more recent approach to object marshalling has been developed and released in prototype form for the Java programming language[30]. This mechanism allows arbitrary Java objects (including active objects, i.e. those that implement the "Runnable" interface) to be communicated through Java I/O facilities (i.e. over network connections, to and from files). The current primary limitation of this approach is that it works only for interpreted byte code Java objects - native code compiled or "just-in-time" compiled objects are not yet supported. This restriction makes the Java mechanism inappropriate for most high performance computing applications. Of course, the other main drawback of this mechanism is that it is limited to programs written in the Java language.

A prototype scheme similar to the Java mechanism but with support for native code objects was implemented for the Emerald distributed system[1]. This scheme, described in [31], requires that native-code versions of Emerald objects be made to periodically reach points in execution (called "bus stops") at which they can be made consistent with their byte-code counterparts. These points are similar in some respects to APrIL poll points. At these points, the existing mechanism for migrating byte-code Emerald objects can be employed. A drawback of this approach is that it is only appropriate for programs written in the Emerald language. A further drawback is the lack of support for programmer specified checkpoint mechanisms for architecture dependent or hand optimized modules.

10 Summary and Future Work

The problem of providing a checkpoint/restart mechanism for use in a heterogeneous environment is significantly complicated when compared to the same problem in a homogeneous environment. On the other hand, the uses for such a mechanism in a heterogeneous environment are at least as important. Although some work has been done to develop an automated mechanism for capturing the state of a process in a platform independent, restartable form, no design has yet been developed that addresses the full set of requirements inherent in a high performance, heterogeneous distributed system. This document proposes a basic design strategy that is being employed to develop a general purpose, flexible, largely automatic heterogeneous checkpoint/restart mechanism appropriate for high performance heterogeneous computing: process introspection. Preliminary experiences in the design, implementation, and performance evaluation of an introspective process checkpointing toolkit have demonstrated that the proposed approach is promising in all respects.

Ongoing work on the Process Introspection project centers on four major elements. First, current efforts are focused on the completion of the design and implementation of the prototype Process Introspection Library (PIL), the APrIL compiler, the Central Checkpoint Coordinator, and checkpointable utility libraries for sequential codes. A significant amount of work in this area remains, including the design and construction of an APrIL compiler to perform the code transformations described in Section 7. The second major goal of ongoing research will be to continue to expand the empirical cost analysis of the checkpoint/restart mechanism for sequential applications. This research is aimed at experimentally determining answers to the following fundamental questions:

- What is the run-time overhead of code transformed to used the PIL as compared to non-transformed code?
- How much does APrIL affect the operation of back-end optimizers? In other words, how great is the speedup of APrIL transformed code with back end optimization as compared to non-transformed, non-checkpointable code?

1. The Emerald scheme actually pre-dates the Java mechanism. The presentation follows this order because the Emerald native code mechanism builds on a general design for object migration of which the Java mechanism is one example.

- What is the observed checkpoint request service latency for APrIL transformed codes? In other words, how long on average and at worst does a process take to begin servicing a checkpoint request given different poll point placement schemes?
- What is the observed cost of performing a checkpoint for processes of different state complexities and sizes?
- What is the observed cost of performing a restart of checkpoints of different complexities and sizes?

The third future research goal will be the integration of introspective checkpointing into at least one parallel distributed system. The general problems associated with integration of the system are somewhat independent of the distributed system targeted for integration. The anticipated integration issues include:

- Checkpointable versions of libraries needed to interact with the system (e.g. message passing) must be constructed. These libraries will likely have inherent operating systems dependencies, and will thus be candidates for hand coded checkpoint mechanisms.
- A version of the CCC based on the intra-process communication interface for the selected system must be constructed.

The fourth major item to be addressed by future work will be the evaluation of the checkpoint mechanism for distributed programs. This evaluation will involve quantitative and qualitative aspects. First, the checkpoint and restart costs of programs in the distributed environment must be measured. This performance will indicate the degree to which using introspective checkpointing is a viable methodology for high performance applications in a real distributed system. Next, some qualitative measurement of the usability of the system in a working environment must be made. This may involve finding users with real programs that need a checkpoint/restart mechanism. The degree to which they can automate the coding of their checkpoint/restart mechanism using the PIL and APrIL will be a direct measure of the ease of use of the system.

References

[1] Y. Artsy and R. Finkel, "Designing a Process Migration Facility: The Charlotte Experience", *IEEE Computer*, pp. 47-56, Sept. 1989.

[2] A. Beguelin, E. Seligman, and M. Starkey, "Dome: Distributed Object Migration Environment", Carnegie Mellon University Technical Report CMU-CS-94-153, May 1994.

[3] K.P. Birman, T.A. Joseph, T. Raeuchle, and A. El Abbadi, "Implementing Fault-Tolerant Distributed Objects", *IEEE Transactions on Software Engineering*, Vol. 11, No. 6, pp. 502-508, June 1985.

[4] M. Bishop and M. Valence, "Process Migration for Heterogeneous Distributed Systems", Dartmouth College Technical Report PCS-TR95-264, Aug. 21, 1995.

[5] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald", *IEEE Transactions on Software Engineering*, Vol. 13, No. 1, pp. 65-76, January 1987.

[6] 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.

[7] S.J. Caughey and S.K. Shrivastava, "Architectural Support for Mobile Objects in Large Scale Distributed Systems", *IWOOOS-95*, Lund, August 1995.

[8] 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.

[9] T.C.K. Chou and J.A. Abraham, "Load Balancing in Distributed Systems", *IEEE Transactions on Software Engineering*, Vol. 8, No. 4, pp. 401-412, July 1982.

[10] F. Douglis and J. Osterhout, "Process Migration in the Sprite Operating System", *Proceedings of the 7th International Conference on Distributed Computing*, pp. 18-25, 1987.

[11] 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.

[12] D.L. Eager, E.D. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. 12, pp. 662-675, May 1986.

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

[14] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

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

[16] 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.

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

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

[19] J. Leon, A.L. Fisher and P. Steenkiste, "Fail-safe PVM: A Portable package for Distributed Programming with Transparent Recovery", Technical Report CMU-CS-93-124, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, February 1993.

[20] 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.

[21] 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.

[22] K. Mandelberg and V.S. Sunderam, “Process Migration in Unix Networks”, *Proceedings of the USENIX Winter Conference*, pp. 357-363, 1988.
[23] A. Nye, *Xlib Programming Manual*, O’Reilly & Associates, 1993.
[24] G. D. Parrington, S. K. Shrivastava, S. M. Wheater and M. C. Little, “The Design and Implementation of Arjuna,” *USENIX Computing Systems Journal*, Vol 8, No 3, 1995.
[25] M.L. Powell and B.P. Miller, “Process Migration in DEMOS/MP”, *Proceedings of the Ninth Symposium on Operating Systems Principles* in *ACM Operating Systems Review*, Vol. 17, No. 5, pp. 110-118, 1983.
[26] 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.
[27] J.M. Smith, “A Survey of Process Migration Mechanisms”, *Operating Systems Review*, Vol. 22, No. 3, pp. 28-40, July, 1988.
[28] P. Smith and N.C. Hutchinson, “Heterogeneous Process Migration: The Tui System”, Technical Report, University of British Columbia, Feb. 28, 1996.
[29] Sun Microsystems, *External Data Representation Reference Manual*, Sun Microsystems, Jan. 1985.
[30] Sun Microsystems, *Java Object Serialization Specification*, Revision 0.9, 1996.
[31] B. Steensgaard and E. Jul, “Object and Native Code Thread Mobility Among Heterogeneous Computers”, SOSP 1995.
[32] V.S. Sunderam, “PVM: A framework for parallel distributed computing,” *Concurrency: Practice and Experience*, vol. 2(4), pp. 315-339, Dec. 1990.
[33] M.M. Theimer, K.A. Lantz, and D.R. Cheriton, “Preemptable Remote Execution Facilities for the V-System”, *Proceedings of the 10th ACM Symposium on Operating System Principles*, Dec. 1985.
[34] 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.
[35] 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.
[36] 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.