

# APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the  
requirements for the degree of  
Doctor of Philosophy, Computer Science



---

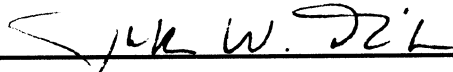
Adam John Ferrari

This dissertation has been read and approved by the Examining Committee:



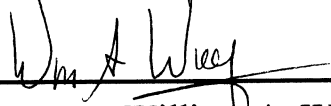
---

Andrew S. Grimshaw - Dissertation Advisor



---

Jack W. Davidson - Committee Chairman



---

William A. Wulf



---

Paul F. Reynolds, Jr.

---

William R. Pearson

Accepted for the School of Engineering and Applied Science:



---

Dean Richard W. Miksad  
School of Engineering and Applied Science

January 1998

**Process State Capture and Recovery  
in High-Performance Heterogeneous Distributed  
Computing Systems**

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

at the

University of Virginia

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy, Computer Science

by

Adam John Ferrari

© Copyright by

Adam John Ferrari

All Rights Reserved

January 1998

# Abstract

Process Introspection is a fundamentally new solution to the process state capture and recovery problem suitable for use in high-performance heterogeneous distributed systems. A process state capture and recovery 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 recoverable on a computer system of a different architecture or operating system platform. The central feature of the Process Introspection approach is automatic transformation of program code to incorporate state capture and recovery functionality. This program modification is performed at a platform-independent intermediate level of code representation, and preserves the original program semantics. The attractive properties of this approach include portability, ease of use, and flexibility with respect to basic performance trade-offs and application-specific requirements. 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. Experimental results obtained using a prototype implementation of the Process Introspection system indicate this mechanism can be applied to computationally demanding scientific applications automatically, resulting in very low run-time overhead (typically below 10%) and efficient state capture and recovery service.

# Acknowledgments

I would like to thank my committee for their guidance, insight, and thoughtful reading of this dissertation. Special thanks to my advisor, Andrew Grimshaw, for believing in this project and for helping me reach this goal. Thanks to Mentat/Legion folks past and present for listening to ideas, coming to practice talks, and for putting up with me at meetings. Thank you to Mike Lewis for a combination of meticulous editing for language, insightful technical criticism, and willingness to read all of the possibly excessive number of pages located beyond this one. Thanks also to Vaidy Sunderam, who first taught me how to do research.

Greatest thanks to Michele Ierardi for understanding about the hundred and one late nights, for encouraging when things were bad, for cheering when things were good, and for liking nerds (well, at least one). Thanks to my parents for always encouraging me to be an individual, and for buying me the Atari 800 on which I hacked my first hundred thousand lines or so. Finally, thanks to Clio for understanding more than most people.

# Contents

<b>Abstract</b>	<b>iv</b>
<b>Acknowledgments</b>	<b>v</b>
<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.1.1 High Performance Heterogeneous Distributed Computing . . . . .	1
1.1.2 Process State Capture and Recovery . . . . .	2
1.2 The Heterogeneous Process State Capture Problem . . . . .	4
1.3 Design Goals . . . . .	6
1.4 Solution Overview . . . . .	9
1.5 Organization . . . . .	11
<b>2 Background and Related Work</b>	<b>12</b>
2.1 High Performance Heterogeneous Distributed Systems . . . . .	12
2.2 The Need for Process State Capture and Recovery . . . . .	15
2.2.1 Load Balancing and Load Sharing . . . . .	15
2.2.2 Fault Tolerance . . . . .	16
2.3 Process State Capture and Recovery . . . . .	17
2.3.1 Homogeneous Systems . . . . .	17
2.3.1.1 Kernel Level Mechanisms . . . . .	17
2.3.1.2 User Level Mechanisms . . . . .	18
2.3.2 Heterogeneous Systems . . . . .	19
<b>3 Process Introspection</b>	<b>23</b>
3.1 System Model . . . . .	23
3.2 Process Model . . . . .	25
3.3 Process Introspection . . . . .	29
3.3.1 State Capture Transformations . . . . .	30
3.3.2 State Recovery Transformations . . . . .	32

3.3.3 Optimizations . . . . .	34
3.4 External State . . . . .	35
<b>4 Correctness Discussion</b>	<b>37</b>
4.1 Heterogeneous Process State Capture Correctness . . . . .	38
4.1.1 General Definition . . . . .	38
4.1.2 Restricted Definition . . . . .	40
4.2 Process Introspection Correctness . . . . .	44
4.2.1 Restrictions . . . . .	47
4.3 Correctness Examples . . . . .	49
4.3.1 Loop Reordering . . . . .	49
4.3.2 Inline Subroutines . . . . .	51
4.3.3 Code Motion . . . . .	53
<b>5 Library Implementation</b>	<b>56</b>
5.1 System Implementation Overview . . . . .	56
5.2 Library Overview . . . . .	58
5.3 Library Implementation . . . . .	60
5.3.1 Buffer Module . . . . .	60
5.3.2 Type Description Table . . . . .	62
5.3.3 Typed Input/Output Module . . . . .	63
5.3.4 Global Variable Table . . . . .	65
5.3.5 Heap Allocation Module . . . . .	66
5.3.6 Pointer Description . . . . .	67
5.3.6.1 Pointer Offset Translation . . . . .	70
5.3.6.2 Pointer Resolution Ordering . . . . .	71
5.3.7 Stack Management . . . . .	72
5.3.8 Code Location Table . . . . .	73
5.3.9 Event Module . . . . .	74
5.3.10 Checkpoint Coordinator . . . . .	74
5.4 System Service Wrappers . . . . .	76

<b>6 The APrIL Source Code Translator</b>	<b>79</b>
6.1 Intermediate Representation . . . . .	79
6.2 APrIL Transformations . . . . .	80
6.2.1 Poll Points . . . . .	81
6.2.1.1 Poll Point Placement Policies . . . . .	83
6.2.2 Function Prologues . . . . .	86
6.2.3 Function Epilogues . . . . .	88
6.2.4 Module Initialization . . . . .	89
6.2.5 Heap Allocation Transformations . . . . .	91
6.2.6 Marshalling Functions . . . . .	93
6.3 Pre-processing . . . . .	94
6.3.1 Variable Declaration Motion . . . . .	94
6.3.2 Function Call Motion . . . . .	95
6.3.3 Function Call Separation . . . . .	97
<b>7 Applications and Performance Results</b>	<b>99</b>
7.1 Performance Metrics . . . . .	99
7.1.1 Performance Overhead . . . . .	99
7.1.2 State Capture and Recovery Costs . . . . .	101
7.2 Experimental Setup . . . . .	103
7.3 Basic Numerical Applications . . . . .	104
7.4 NAS Benchmark Kernels . . . . .	111
7.5 Environmental Simulation . . . . .	116
7.6 Biological Sequence Comparison . . . . .	120
7.7 Performance Discussion . . . . .	125
<b>8 Extensions</b>	<b>127</b>
8.1 Additional Programming Languages . . . . .	127
8.1.1 Fortran . . . . .	130
8.1.2 C++ . . . . .	135
8.2 Supporting Threads . . . . .	141



<b>9 Conclusions and Future Directions</b>	<b>150</b>
9.1 Contributions .....	150
9.2 Future Work .....	151
<b>Appendix A: Performance Data</b>	<b>155</b>
A.1 Basic Numerical Applications .....	156
A.2 NAS Benchmarks .....	166
A.3 Environmental Simulation .....	174
A.4 Biological Sequence Comparison .....	176

# List of Figures

Figure 1.1 Heterogeneous process state capture and recovery. . . . .	5
Figure 3.1 A metasystem. . . . .	24
Figure 3.2 The program model. . . . .	26
Figure 3.3 Program execution on the IR virtual machine. . . . .	29
Figure 3.4 Creation of an introspective program. . . . .	30
Figure 4.1 Equivalent continuous executions of a program. . . . .	40
Figure 4.2 Consistency points. . . . .	43
Figure 4.3 Axy loop. . . . .	49
Figure 4.4 Axy loop, transformed. . . . .	50
Figure 4.5 Axy invocation, transformed. . . . .	51
Figure 4.6 Code motion. . . . .	53
Figure 4.7 Transformed code motion example. . . . .	54
Figure 4.8 Transformed example, code motion applied. . . . .	55
Figure 5.1 Using the Process Introspection system. . . . .	58
Figure 5.2 PIL modules and dependencies. . . . .	60
Figure 5.3 PIL buffer usage example. . . . .	62
Figure 5.4 Type registration examples. . . . .	63
Figure 5.5 Typed I/O interface. . . . .	64
Figure 5.6 User defined marshalling function registration. . . . .	64
Figure 5.7 Global variable registration. . . . .	65
Figure 5.8 PIL heap management example. . . . .	67
Figure 5.9 Pointer description interface. . . . .	68
Figure 5.10 Pointer description. . . . .	69
Figure 5.11 Pointer offset translation algorithm. . . . .	71
Figure 5.11 Local variable registration example. . . . .	72
Figure 5.12 Explicit stack save/restore interface examples. . . . .	73
Figure 5.13 Function pointer registration. . . . .	74
Figure 5.14 Process Introspection file interface usage. . . . .	76
Figure 6.1 An optional poll point. . . . .	82

Figure 6.2 A mandatory poll point.....	83
Figure 6.3 A function prologue. transformation.....	88
Figure 6.4 A function epilogue. ....	89
Figure 6.5 Heap management expression grammar.....	91
Figure 6.6 Type allocation heuristic examples. ....	92
Figure 6.7 Function call motion, <b>if</b> statement example.....	96
Figure 6.8 Function call motion, <b>for</b> loop example.....	96
Figure 6.9 Function call separation example.....	98
Figure 8.1 Alternate language support using the existing implementation. ....	128
Figure 8.2 Alternate language support using a universal IR. ....	129
Figure 8.3 Alternate language support using source-to-source translation. ....	130
Figure 8.4 An optional poll point in Fortran. ....	130
Figure 8.5 A mandatory poll point in Fortran. ....	131
Figure 8.6 A function prologue transformation in Fortran. ....	132
Figure 8.7 Registration of variables in common blocks. ....	135
Figure 8.8 C++ constructor/destructor transformations.....	137
Figure 8.9 Wrapper <b>new</b> operator registration. ....	138
Figure 8.10 A template function. ....	139
Figure 8.11 Overloaded type number functions. ....	139
Figure 8.12 C++ exception handling example.....	141
Figure 8.13 A simple threads interface. ....	143
Figure 8.14 Wrapper semaphore record.....	147
Figure 8.15 Wrapper semaphore P operation. ....	147
Figure 8.16 Wrapper semaphore V operation. ....	148

# List of Tables

Table 7.1 Test platforms. . . . .	103
Table 7.2 Basic applications, average $O_{norm}$ . . . . .	105
Table 7.3 Basic applications, optimizer effectiveness. . . . .	106
Table 7.4 Basic applications, average $O_{opt}$ . . . . .	107
Table 7.5 Basic applications, per-platform average $O_{opt}$ . . . . .	108
Table 7.6 Basic applications, poll point counts. . . . .	108
Table 7.7 Basic applications, average poll point intervals on x86. . . . .	109
Table 7.8 Basic applications, time to checkpoint/restart on x86. . . . .	110
Table 7.9 NAS benchmarks, average $O_{norm}$ . . . . .	112
Table 7.10 NAS benchmarks, optimizer effectiveness. . . . .	113
Table 7.11 NAS benchmarks, average $O_{opt}$ . . . . .	113
Table 7.12 NAS benchmarks, poll point counts. . . . .	115
Table 7.13 NAS benchmarks, average poll point intervals on x86. . . . .	115
Table 7.14 NAS benchmarks, time to checkpoint/restart on x86. . . . .	116
Table 7.15 LAI, overhead and optimizer effectiveness. . . . .	117
Table 7.16 LAI, poll point counts and average poll point intervals. . . . .	120
Table 7.17 LAI, checkpoint/restart costs. . . . .	120
Table 7.18 FASTA, overhead and optimizer effectiveness. . . . .	122
Table 7.19 Smith-Waterman, overhead and optimizer effectiveness. . . . .	122
Table 7.20 FASTA, poll point counts and average poll point intervals. . . . .	123
Table 7.21 Smith-Waterman, poll point counts and average poll point intervals. . . . .	123
Table 7.22 FASTA, time to checkpoint/restart. . . . .	124
Table 7.23 Smith-Waterman, time to checkpoint/restart. . . . .	124
Table 7.24 Performance with and without marshalling functions. . . . .	125
Table A.1 Matrix multiply, execution times (seconds). . . . .	156
Table A.2 Matrix multiply, overhead and optimizer effectiveness. . . . .	156
Table A.3 Matrix multiply, poll point counts. . . . .	157
Table A.4 Matrix multiply, average poll point interval (milliseconds). . . . .	157
Table A.5 Matrix multiply, time to checkpoint/restart (milliseconds). . . . .	157

Table A.6 Gauss-Seidel, execution times (seconds). . . . .	158
Table A.7 Gauss-Seidel, overhead and optimizer effectiveness. . . . .	158
Table A.8 Gauss-Seidel, poll point counts. . . . .	159
Table A.9 Gauss-Seidel, average poll point interval (milliseconds). . . . .	159
Table A.10 Gauss-Seidel, time to checkpoint/restart (milliseconds).. . . . .	159
Table A.11 Quicksort, execution times (seconds).. . . . .	160
Table A.12 Quicksort, overhead and optimizer effectiveness.. . . . .	160
Table A.13 Quicksort, poll point counts. . . . .	161
Table A.14 Quicksort, average poll point interval (milliseconds).. . . . .	161
Table A.15 Quicksort, time to checkpoint/restart (milliseconds). . . . .	161
Table A.16 Gaussian elimination, execution times (seconds).. . . . .	162
Table A.17 Gaussian elimination, overhead and optimizer effectiveness.. . . . .	162
Table A.18 Gaussian elimination, poll point counts. . . . .	163
Table A.19 Gaussian elimination, average poll point interval (milliseconds).. . . . .	163
Table A.20 Gaussian elimination, time to checkpoint/restart (milliseconds). . . . .	163
Table A.21 Conjugate gradient, execution times (seconds). . . . .	164
Table A.22 Conjugate gradient, overhead and optimizer effectiveness. . . . .	164
Table A.23 Conjugate gradient, poll point counts.. . . . .	165
Table A.24 Conjugate gradient, average poll point interval (milliseconds). . . . .	165
Table A.25 Conjugate gradient, time to checkpoint/restart (milliseconds).. . . . .	165
Table A.26 NAS IS, execution times (seconds). . . . .	166
Table A.27 NAS IS, overhead and optimizer effectiveness. . . . .	166
Table A.28 NAS IS, poll point counts.. . . . .	167
Table A.29 NAS IS, average poll point interval (milliseconds). . . . .	167
Table A.30 NAS IS, time to checkpoint/restart (milliseconds).. . . . .	167
Table A.31 NAS EP, execution times (seconds).. . . . .	168
Table A.32 NAS EP, overhead and optimizer effectiveness.. . . . .	168
Table A.33 NAS EP, poll point counts. . . . .	169
Table A.34 NAS EP, average poll point interval (milliseconds).. . . . .	169
Table A.35 NAS EP, time to checkpoint/restart (milliseconds). . . . .	169
Table A.36 NAS MG, execution times (seconds). . . . .	170

Table A.37 NAS MG, overhead and optimizer effectiveness. . . . .	170
Table A.38 NAS MG, poll point counts. . . . .	171
Table A.39 NAS MG, average poll point interval (milliseconds). . . . .	171
Table A.40 NAS MG, time to checkpoint/restart (milliseconds). . . . .	171
Table A.41 NAS CG, execution times (seconds). . . . .	172
Table A.42 NAS CG, overhead and optimizer effectiveness. . . . .	172
Table A.43 NAS CG, poll point counts. . . . .	173
Table A.44 NAS CG, average poll point interval (milliseconds). . . . .	173
Table A.45 NAS CG, time to checkpoint/restart (milliseconds). . . . .	173
Table A.46 LAI, execution times (seconds). . . . .	174
Table A.47 LAI, overhead and optimizer effectiveness. . . . .	174
Table A.48 LAI, poll point counts. . . . .	175
Table A.49 LAI, average poll point interval (milliseconds). . . . .	175
Table A.50 LAI, time to checkpoint/restart (milliseconds). . . . .	175
Table A.51 FASTA, execution times (seconds). . . . .	176
Table A.52 FASTA, overhead and optimizer effectiveness. . . . .	176
Table A.53 FASTA, poll point counts. . . . .	177
Table A.54 FASTA, average poll point interval (milliseconds). . . . .	177
Table A.55 FASTA, time to checkpoint/restart (milliseconds). . . . .	177
Table A.56 Smith-Waterman, execution times (seconds). . . . .	178
Table A.57 Smith-Waterman, overhead and optimizer effectiveness. . . . .	178
Table A.58 Smith-Waterman, poll point counts. . . . .	179
Table A.59 Smith-Waterman, average poll point interval (milliseconds). . . . .	179
Table A.60 Smith-Waterman, time to checkpoint/restart (milliseconds). . . . .	179

# Chapter 1

## Introduction

This dissertation describes the design, implementation, and empirical analysis of a fundamentally new approach to the problem of capturing the dynamic state of a process in a platform-independent form, and then later recovering that process state in a semantics-preserving manner, possibly on a different type of host computer system. This new approach, called Process Introspection, centers on the semantics-preserving, typically automatic transformation of programs to incorporate autonomous state capture and recovery functionality. The attractive properties of Process Introspection include portability, ease of use by the programmer, flexibility with respect to basic performance trade-offs, and applicability in a variety of systems contexts and for a wide range of applications with distinct requirements. Our solution is novel in its true platform and run-time-system independence—our core process state capture and recovery mechanisms require no system support or non-portable code. Furthermore, Process Introspection is the first heterogeneous process state capture and recovery mechanism to have demonstrably low impact on performance.

### 1.1 Motivation

#### 1.1.1 High Performance Heterogeneous Distributed Computing

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 [1, 34, 38, 41]. 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 distributed systems and parallel computers. The complexity of varying architectural features such as data rep-

resentation and instruction sets, and varying operating system features such as process management, communication, and file system interfaces must be masked from the application programmer. Heterogeneity also complicates existing problems in parallel and distributed systems. For example, task placement may depend on processor speed and type, the set of operating system services that are available on nodes, 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 further improve performance by mapping sub-tasks of a computation to the most appropriate available hardware (called *superconcurrency* by Freund and Cornwell [32]) makes heterogeneous computing a promising area of research.

### **1.1.2 Process State Capture and Recovery**

Early experiences with metacomputing systems and applications have demonstrated the need for a process state capture and recovery mechanism—a mechanism to automatically checkpoint the state of a running program in some stable form and then later restart the program from the point of capture, possibly on a different host. A substantial body of research has already demonstrated the utility and desirability of such a mechanism in homogeneous environments. For example, process migration policies supporting load sharing and/or fault tolerance can be based on a process state capture facility (e.g. Condor [54]). Process state capture and recovery is also the basis of a large class of backward error recovery schemes documented in the fault tolerance literature [24]. Beyond failure resilience, the ability to “roll back” a computation afforded by a process state capture and recovery mechanism can be used to implement the semantics of certain programming environments. For example, systems such as Time Warp [43] and Hope [19] rely on process state capture and recovery to provide semantic guarantees such as the causal ordering of message deliv-



ery to a process.

Well-documented uses for a process state capture and recovery mechanism such as load sharing and fault tolerance address issues of increasing importance in metacomputing environments. The inherent variety of processor capabilities and the often unpredictable loads in such systems due to resource sharing and varying quality-of-service policies make the migration of computationally demanding processes an imperative. Similarly, the growing scale of metasystems and the applications that run on them has increased the likelihood that one or more components of a program will be subject to node failure, thus strengthening the argument for increased employment of fault tolerance techniques. However, beyond these existing uses for process state capture and recovery, the availability of a such a mechanism in the metasystem context also opens up new possibilities. For example, a metasystem such as Legion [38] could use process state capture and recovery for improved resource management; if the number of active entities in the system became greater than could be efficiently supported, the system could temporarily preempt the execution of some jobs by checkpointing and terminating their processes, then later restarting them when the system load became lighter<sup>1</sup>. 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. The increasing importance of existing uses of process state capture and recovery, together with the new possibilities introduced by metasystems, has made the design of a such a mechanism a key research issue in metacomputing [74].

Whereas 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 absent from most existing heterogeneous computing systems; because of the additional inherent complexity introduced by heterogeneity, few designs for such a facility have been developed to date. In homogeneous

---

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

systems, process state capture and recovery 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 straightforward to capture. 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 layout make the raw process state incompatible across different platforms.

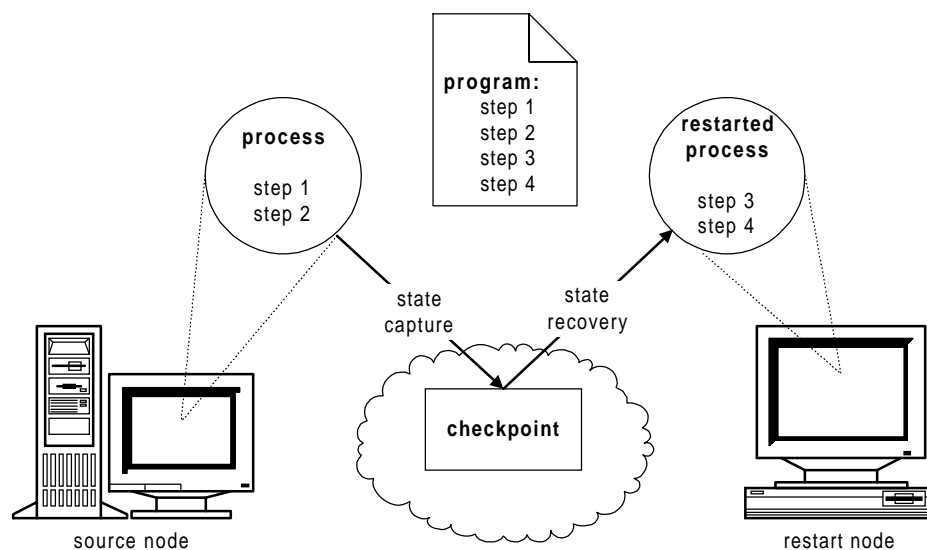
In a heterogeneous environment, the state of a process cannot be captured using the naive approach that suffices in the homogeneous case. To mask the varying features of a migrating process's environment in a heterogeneous system, a state capture mechanism must examine and capture the logical internal structure and meaning of the process's address space contents as well as any external operating system specific information such as file system or intra-process communication state. This prospect is somewhat daunting—the logical point in execution, the call stack (or call stacks, if threads are supported), complex data structures, the logical structure and contents of heap allocated memory, and all other process state must be analyzed and captured in a platform-independent format, masking data format differences, addressing differences, instruction set differences, and so on.

## **1.2 The Heterogeneous Process State Capture Problem**

In this dissertation we describe the design and implementation of an effective, general, and portable solution to the heterogeneous process state capture and recovery problem. Before doing

so, we must more precisely delineate this problem.

A mechanism solving the heterogeneous process state capture and recovery problem must provide the ability to generate a checkpoint for an active process—a complete description of that process’s state and point in execution. The mechanism must also support the later use of that checkpoint to restart a process with equivalent state and at an equivalent point in execution, possibly on a different type of computer from the one on which the original the checkpoint was created. Figure 1.1 depicts the basic operation of a heterogeneous state capture and recovery mechanism. The possibility of cross-platform restarts leads to the most fundamental solution constraint: the mechanism should generate *platform-independent* checkpoints—i.e. checkpoints produced on a computer system of any architecture and operating system should be recoverable on a system of any other architecture and operating system.



**Figure 1.1:** Heterogeneous process state capture and recovery.

As stated thus far, we wish to design a mechanism to capture the state of a running program on one type of computer, and restart that program from an equivalent point in execution on a different type of computer. This general problem statement permits a wide range of possible solutions. For example, one straightforward approach is to use an interpreted language, as in mobile

agent systems such as Sumatra [1] and Ara [67,68]. In these designs, the interpreter acts as a virtual machine that can artificially homogenize a system composed of heterogeneous elements.

Whereas these solutions are appropriate for typical agent-based applications such as information retrieval, in this work we wish to address the requirements of a different application domain: computationally demanding scientific applications—programs for which the requirement of an interpretive execution environment would lead to unacceptable performance degradation. The target application domain for our mechanism includes, but is not limited to, computationally demanding sequential applications and high-performance distributed memory parallelized scientific programs exhibiting medium to coarse granularity.

Given this target application set, the intended environment for our solution is a high-performance distributed system consisting of a variety of node computing systems: a *metasystem*. The 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 local operating systems, run on the nodes, typically executing in native code form due to performance considerations. Systems that experience node failures and that exhibit load imbalance among the nodes, for example due to resource sharing, are of primary interest.

### 1.3 Design Goals

Based on our statement of the heterogeneous process state capture and recovery problem, and on our target environment and application area, we state the following design goals for our solution:

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

migrate a program running in an embedded system controlling a robot to a general-purpose workstation automatically without special programmer effort to implement an emulation of the program's robot environment on the workstation). A large subset of problems of interest in high-performance computing can be solved using platform-independent programs—evidence of this fact includes the large number of scientific numerical applications that are implemented in Fortran and that are basically portable across standard Fortran implementations.

In some cases, certain modules of a program will inherently rely on platform-specific functionality. For example, many programs rely on the file system interface of the environment in which they execute. In other cases, certain program modules will not be best expressed using a platform-independent specification. For example, given the well-known observation that most programs spend 90% of their execution time in less than 10% of their program text, it is often desirable to implement certain critical sections of a program in carefully hand-optimized assembly language. Since certain modules will need to be expressed using platform-dependent specifications, a convenient user interface should be provided for incorporating state capture and recovery support into architecture-dependent modules in a manner that is interoperable with automatically supported modules. This interface should give the programmer flexibility to customize and tune a module's state capture and recovery mechanism when appropriate; at the same time, it should be easy to use.

- **Efficiency**—Ideally, we would like a heterogeneous state capture and recovery mechanism to perform with comparable costs to a checkpoint mechanism designed for a homogeneous system. 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 dump of the process. Naturally, heterogeneity will add some cost, but as a goal the heterogeneous mechanism should remain comparable to the base-case costs of a homogeneous mechanism.

In addition to providing efficient state capture and recovery, the mechanism should introduce low run-time execution speed overhead (ideally, no overhead should be introduced). In particular, if checkpoints are not performed during a certain period of execution, a process with state capture and recovery service available to it should not run significantly slower than an optimized version of the code without this service available over the same period. This goal might be stated simply as, “Don’t pay if you don’t play.” Because Process Introspection is based on the modification of application programs, this design goal is especially important in our solution. Whereas our mechanism will definitely introduce a certain amount of overhead, the degree to which this introduced overhead can be controlled is of particular interest. As a design goal, we wish to achieve a net introduced overhead of less than 10%.

- **Responsiveness**—The state capture mechanism should provide low checkpoint-request delay—i.e. the time between a when checkpoint is scheduled or requested and when the checkpoint begins to be produced should be significantly less than the time required to produce 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).
- **Generality**—The mechanism should be appropriate for use with a wide variety of programs that are written in a variety of languages, and that solve a wide range of problems. This precludes special purpose toolkits appropriate only for problems of a certain structure such as Dome [7], which provides automatic state capture and recovery for array data types suitable for use in data-parallel applications. Furthermore, the mechanism should be adaptable to a variety of different heterogeneous environments (e.g., a PVM environment [34], a Condor-like system [54], etc.).

## 1.4 Solution Overview

The Process Introspection solution to the heterogeneous process state capture and recovery problem is based on the premise that programs can be modified to incorporate their own checkpoint and restart functionality. Put simply, programs are modified to be both self-describing and self-recovering. Furthermore, for program modules that are expressed in a platform-independent form, the code modifications required to support Process Introspection are performed automatically by a compiler.

The most basic Process Introspection program modifications enable a process to capture the state of any data structure in its address space automatically and in a platform-independent form, and to recover data structures stored in this platform-independent form. These modifications involve the addition of code to maintain a record of the size and type of all memory regions used by the process. Furthermore, a mechanism based on these memory-region descriptions is added to the process to allow the automatic production of a logical, platform-independent description of any data region, and conversely to allow the automatic recovery (i.e. re-allocation and re-initialization) of any memory region based on its logical description. This level of program modification addresses issues such as data-format and address space structure differences, but does not address the capture and recovery of a process's point in execution.

The capture and recovery of a process's execution environments (i.e. call stacks and threads) are supported through a specialized use of subroutine call and return mechanisms. The subroutine return mechanism is used to traverse a process's invocation stacks during state capture, and the subroutine call mechanism is used to artificially reconstruct the stacks during state recovery. The code modifications performed in order to support this specialized use of subroutine call and return mechanisms involve the introduction of *poll points* and *subroutine prologues*.

Poll points are introduced points in execution at which the process determines if a checkpoint should be produced (analogous to Bus Stops in Heterogeneous Emerald [82]). At a poll point, if

the process determines that a checkpoint should be produced, code is executed to save the state of the current active subroutine and to return immediately to the calling subroutine. Poll points are placed throughout the program in a manner that ensures that after any subroutine return during a state capture operation, a poll point is encountered immediately. At this next poll point, the state of the calling subroutine is saved, the calling subroutine immediately returns to its caller, and the process is repeated until the base subroutine activation is reached.

Similarly, to effect restarts, the process employs subroutine prologues. When a process state recovery is initiated, the base subroutine is activated and executes an added prologue that restores its local state. This prologue then jumps to the appropriate location in the subroutine, which will be a call to the next subroutine in the checkpointed stack. When activated, this subroutine recovers its state, jumps to the appropriate location in the code, and the process repeats itself until the stack is completely restored.

This method of process state capture and recovery uses only common features of high-level procedural languages, and certainly could be employed directly by a programmer. The problem with applying 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, as will be discussed in Chapter 6, for many programs these modifications can be performed automatically by a source code translator.

The thesis of this dissertation is that Process Introspection is an effective, correct, efficient solution to the heterogeneous process state capture and recovery problem. The general strategy of Process Introspection, discussed at a high level in this section, and in greater detail in Chapter 3, can be used as the basis for a useful, automatic, portable, and efficient implementation of a heterogeneous process state capture and recovery mechanism. This mechanism can be applied to non-



trivial, high-performance applications, for which it can provide efficient state capture and recovery service while introducing very low performance overhead, typically less than a 10% slowdown over standard, optimized native-code executables.

## **1.5 Organization**

The rest of this dissertation is structured as follows to support this thesis: In Chapter 2 we discuss background and related work. In Chapter 3 we describe the Process Introspection model and fundamental algorithms in detail. In Chapter 4 we discuss the correctness of the proposed process state capture and recovery algorithms. In Chapter 5, we discuss the implementation of a run-time library to support the application of the Process Introspection methodology. In Chapter 6, we describe the implementation of a source code translation tool that automatically applies the Process Introspection methodology to platform-independent programs written in ANSI C. In Chapter 7, we describe application performance experiments using the prototype system implementation. In Chapter 8, we discuss the design of extensions to the current system implementation, including support for additional programming languages, and support for programming constructs such as threads. Finally, in Chapter 9, we present conclusions and discuss future work on this research.

# Chapter 2

## Background and Related Work

Before discussing Process Introspection in greater detail, we first examine background and related work in the area of process state capture and recovery in heterogeneous computing environments. We begin by examining general background work in high performance heterogeneous computing. We then examine motivating applications for a process state capture and recovery mechanism. Finally, we examine past and current research directly related to the process state capture and recovery problem, concentrating on systems and designs that support heterogeneity.

### 2.1 High Performance Heterogeneous Distributed Systems

Over the past decade, the use of heterogeneous collections of computing systems interconnected by one or more networks as a single logical computational resource has grown. This technique, referred to variously as *network heterogeneous computing* [25], *mixed-machine heterogeneous computing* [74], or *metacomputing* [38, 75] is a promising area of research for a variety of reasons. For example, in a metacomputing environment, tasks can be moved to the computing resources best suited for their needs during different phases of computation, an idea termed *superconcurrency* in Freund and Cornwell [32]. Furthermore, metacomputing promises to enable the use of unprecedented amounts of computational power for individual applications. The evolution of computing technology and economics over time results in heterogeneity among the available resources. Even within a single institution, the set of resources available to a user is almost always heterogeneous. Metacomputing allows individual applications to harness the aggregate power of these inherently heterogeneous collections of resources.

The hardware makeup of a metacomputing system can vary widely, ranging from local area networks such as Ethernet or FDDI connecting personal computers, to high performance networks

such as ATM or Myrinet connecting high-end workstations and MPPs. The key distinguishing feature of a metacomputing environment is software. Metacomputing software is responsible for providing a unified programming and resource management interface to a complex heterogeneous collection of hardware resources. Although numerous software systems that support some form of network heterogeneous computing have been documented and/or are available [88], the majority of use has thus far been based on a small set of popular, simple packages such as Parallel Virtual Machine (PVM) [81, 34], and the Message Passing Interface (MPI) [41].

PVM provides the programmer with a library interface that supports an explicit message-passing, distributed memory MIMD programming model. Library routines are provided to create new tasks, marshal vectors of basic data types into buffers, and pass message buffers between tasks asynchronously. In addition to the library programming interface, the PVM software system provides tools for specifying and managing a virtual machine on which applications will execute.

Similar to PVM, the Message Passing Interface supports an explicit message passing, MIMD programming model through a library interface. In its most recent version (MPI-2 [61]), MPI adds to the PVM level of functionality features such as a uniform parallel I/O interface, collective communication operations (library calls that specify communication patterns between more than two tasks, e.g. all-to-all), and one sided communication (or remote memory access, the ability of one task to asynchronously read or write another task's address space).

Whereas MPI, PVM, and the large class of systems that is essentially isomorphic to these [88] provide essentially identical message-passing based programming models, a few network heterogeneous systems have supported alternative, higher-level models. For example, Linda [13] provides a shared tuple space into which tasks can read and write data in order to cooperate with one another. Mentat [37] supports a C++-based programming model in which the basic units of computation and scheduling are active objects that communicate via asynchronous methods. In Mentat, arguments to methods are passed according to their data-dependence graph, thereby

minimizing communication. Furthermore, processes do not block for method results until those results are needed, a policy that increases concurrency. This model of computation is known as *macro-dataflow*.

MPI, PVM, and other first-generation metacomputing systems provide a relatively low-level programming and resource management interface, and thus have been employed most successfully in simple, small-scale environments. Recently, a number of projects have begun to examine the issues involved in larger-scale metacomputing. One such project, Globus [31], provides support for a number of pre-existing parallel computing interfaces (such as MPI) in a metacomputing environment with enhanced resource management tools, security features, and a wide-area-aware communication system, Nexus [29]. Similar “collection and integration of separate services” architectures are employed in a number other metacomputing systems such as the Berkeley NOW project [2], MOL [71], and I-Soft [30]. An alternative approach to metacomputing employed by the Legion project [39, 40] is based on distributed active objects. In Legion, all system and application software components are active objects with logically disjoint address spaces [52]. These objects interact via non-blocking, data-driven method invocations, in a manner based on the underlying concepts of Mentat. A primary advantage of the Legion object-based approach is the flexibility to modify or replace both system and application components using a single underlying object model, programming interface, and tool set. Systems such as Legion and Globus promise to increase both the scale and availability of metacomputing, and further increase the need for research in metacomputing issues.

Although metasystems software is not yet mature, early results on heterogeneous network computing testbeds have been promising. For example, a detailed performance study of the NAS benchmark suite [5] using PVM on the HEAT testbed [51] was presented by White, Alund, and Sunderam [93]. Their results showed that relatively small clusters of workstations could provide performance within an order of magnitude of a 1-node Cray Y-MP. The Legion group has reported

performance results obtained on a wider area, more significantly heterogeneous campus-wide testbed [40]. Using a biological sequencing application [66], they found that a 64 node heterogeneous network of workstations was able to perform as well as a 32-node Intel Paragon. These experiences are just two examples of the growing documented evidence of the excellent results achievable through metacomputing [60].

Applications experiences have lead metacomputing practitioners to a number of general observations about heterogeneous computing. First, although metacomputing can be beneficial for both computationally intensive jobs and short-running interactive jobs (e.g. through remote execution and load sharing), a key to the effective use of metacomputing environments is the management of large, computationally expensive, possibly parallel applications [3]. Secondly, effective parallel programs in a heterogeneous environment will almost always exhibit coarse granularity, that is, a high amount of computation performed for each byte of information communicated between tasks [93].

## **2.2 The Need for Process State Capture and Recovery**

Whereas the ability to capture and restore the state of a process has been available for some time in a variety of homogeneous distributed systems [4, 20, 54, 64, 70, 86], this feature has been absent in most metacomputing environments. This is due primarily to the difficulty of implementing such a feature, not due to a lack of need for such a mechanism.

### **2.2.1 Load Balancing and Load Sharing**

It has long been recognized that adaptive load sharing as enabled by a process state capture and recovery mechanism is an effective means of increasing performance (e.g. throughput, response time, etc.) in a distributed system [22, 63, 73]. Systems such as NOW [1] and Condor [54] rely on the ability to capture the state of a process on one node and resume that pro-

cess on a different node to achieve more equitable distributions of work among the nodes in a metasytem. A performance study performed on the Berkeley NOW system [3] determined that not only was adaptive load sharing effective at increasing the performance of both parallel and sequential applications in a shared environment, but the reduction of impact on interactive users by parallel jobs afforded by load sharing was critical to the social acceptance of parallel computing in a shared network system.

The issue of adaptive load sharing is likely more critical in metasytems as compared to homogeneous distributed systems. Metasytems suffer an increased likelihood of load-imbalance, even when used in a dedicated mode for parallel applications, due to the varying performance of heterogeneous nodes. Added to this is the shared nature of many metasytems, where interactive users and long-running parallel applications must compete for resources.

### **2.2.2 Fault Tolerance**

The ability to capture and later recover the full state of a process is the basis of a large class of backward error recovery techniques for fault tolerance [24]. For example, systems such as Fail-safe PVM [50] rely on this ability. The use of process checkpointing for rollback-recovery protocols typically relies on the ability either to synchronize the checkpoints of cooperating tasks (for example, using the checkpoint coordination algorithms of Chandy and Lamport [18] or Mattern [59]), or to propagate the rollback of related tasks to reach a consistent state [89]. However, the development of such coordination and propagation schemes is orthogonal to the heterogeneity of the computing nodes involved, and thus we can concentrate on the issue of process state capture and recovery confident that existing higher-level application checkpoint/restart techniques will remain applicable in heterogeneous environments. This observation is important—the need for fault tolerance in metasytems is significant and is growing with the scale of those systems. In, For example, the Legion group has reported observed fault rates as poor as one node failure per

hour on a 64-node campus-wide metasystem testbed [40].

## 2.3 Process State Capture and Recovery

Thus far, we have examined background work in metacomputing systems, and motivated the need for a process state capture mechanism in such environments. We now focus on directly related work—designs and implementations of systems that support process state capture and recovery. Before considering mechanisms intended for use in heterogeneous systems, it is worth examining the existing mechanisms designed for use in homogeneous environments.

### 2.3.1 Homogeneous Systems

#### 2.3.1.1 Kernel Level Mechanisms

Process state capture mechanisms to support activities such as process migration and checkpoint/restart have been the subject of a great deal of research, both in terms of mechanisms and policies [62, 64, 76]. Most homogeneous state capture mechanisms are implemented inside operating systems at the kernel level due to efficiency concerns and because a process's external state is more readily available at that level. For example, systems such as Charlotte [4], Sprite [20], DEMOS/MP [70], and the V-System [86] utilize kernel-level state capture and recovery mechanisms to support process migration.

Although these and other kernel-level homogeneous state capture mechanisms differ in certain performance-related respects, they share a common basic approach to capturing the state of a process. The state of the process is commonly defined to consist of [26]:

- *virtual memory*—code, stack, and data segments of the process's address space
- *open files*—file descriptors, file pointers, I/O buffers, etc.
- *communication buffers*—connection information, message buffer contents, etc.
- *processor state*—current condition codes, program counter, stack pointer, general purpose registers, etc.

- *environment data*—process identifier, user name, etc.

All of these parts of the process state are accessible at the kernel level, and thus state capture involves marshalling or communicating this information in a well-defined format. For example, capturing a process's virtual memory might involve saving the page table of the process along with the contents of any valid pages. Of course, the implementation of state capture operations varies widely depending on the intended use and context. For example, during process migration, an effort is often made to transfer the minimal state needed to restart the process at its destination first, and to transfer remaining state subsequently to reduce migration latency [95]. Alternatively, for the purposes of checkpointing, incremental schemes for saving a process's memory, such as periodically capturing only dirty pages, may improve time/space performance [69].

Beyond the obvious issue of heterogeneity, kernel-level state capture schemes have a number of undesirable features in metasytem contexts. 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 metasytem approaches, it is typically infeasible to mandate replacement of the operating system on all participating nodes. These issues, along with the requirement of support for heterogeneity, strongly suggest the use of user-level state capture mechanisms.

### **2.3.1.2 User Level Mechanisms**

A number of systems to date have provided some form of homogeneous process state capture implemented at the user level (i.e. without direct, special kernel support) [14, 55, 54, 69]. For example, Condor [55] performs process state capture and recovery in homogeneous environments by using a slightly modified core dump of the process to capture and recover memory and processor state. 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 via wrapper routines. An alternative approach described by Plank et. al. [69] links programs with a special library



that contains code to capture a process's internal state. In this design, processor state is captured using the Unix `setjmp` system call.

Although these approaches are typically somewhat less efficient than kernel level implementations, user-space designs are generally more portable (e.g. Condor and libchckpt [69] are highly portable among Unix-based platform). A common argument against user-level state capture schemes is the difficulty involved in capturing and recovering a process's external and kernel-level state. For example, some user-level approaches such as Condor and Mandelberg/Sunderam [57] restrict certain forms of intra-process communication mechanisms. However, this argument against user-level state capture mechanisms is largely unfounded, as demonstrated systems such as MIST/MPVM [15, 16], Fail-safe PVM [50], and Hector [72]. These systems provide a location-independent communication layer that renders process migrations transparent to message passing operations performed at the application level. Similar user-level wrappers are possible for other services that involve external state such as file systems.

### 2.3.2 Heterogeneous Systems

Thus far, we have only considered systems that perform state capture and recovery in homogeneous environments. However, 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 also been the subject of a significant amount of prior work. Perhaps the most general coverage of this topic is presented by von Bank, Shub, and Sebesta [90], who developed the idea that a procedural computation can be modeled as progression through a sequence of compatible well-defined states: points in execution at which the state of a process can be used to fully describe the equivalent state of any other implementation of the process. In our model, these compatible well-defined states occur when poll points are encountered. Related implementation work done by this group integrated a limited form of heterogeneous process migration into the V system [21].

This implementation relied on the operating system to examine and translate the memory state of the process based on type information embedded in the process's program text. The design suffered certain limitations, particularly the requirement that data structures reside at identical addresses in all versions of a process. In truly heterogeneous systems, this restriction could not be observed in general, and thus mobility between some nodes would be impossible.

A novel approach to the heterogeneous state capture/restore problem was proposed by Theimer and Hayes [87]. 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 to proceed from its logical point of state capture. The actual process migration then entails 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)—essentially, the process effects its own restart. 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 [82], who developed 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, plus heterogeneous mobility for interpreted byte-code programs). 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 modifications to the generated code are 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—whereas we require modification of programs to incorporate state capture and recovery mechanisms, we do not require support from any external agents. This affords us the desirable attribute of generality—our tool can be integrated into existing distributed systems without requiring modification to those systems or to our basic process state capture mechanism, and Process Introspection does not require extensive run-time system support. Our current implementation requires only that the system interface be accessible from C code, and that it be possible to construct a wrapper interface for system services that maintain external state for processes.

A similar approach to that of heterogeneous Emerald, called Tui [77], 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, as opposed to the Emerald approach which mandates the use of the less common Emerald language. 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.

The growing interest in the area of mobile agents has resulted in a number of state-capture and recovery mechanisms to support migration in mobile agent languages. For example, the Sumatra [1] language supports the capture and recovery of Java threads in a heterogeneous environment. State capture and recovery in Sumatra is implemented by a set of modifications to the Java Virtual Machine [35] bytecode interpreter. A more flexible approach is supported by the Ara system [67, 68]. As opposed to Sumatra, which mandates use of the Java language, Ara supports mobile agents in an extensible set of interpreted languages, currently including interpreted C and

Tcl [65]. To support state capture of a running agent, the interpreters used in the system must be able to capture their own full state (i.e. including the state of the program being interpreted). A primary drawback of these and other mobile agent systems such as TACOMA [44,45], Agent Tcl [36], and Telescript [92] is the use of interpreted execution for agents. In our intended application domain, this model fails to meet the performance requirements of most users.

One system that overcomes this limitation is Extended Facile [48], an agent programming system based on the Facile functional programming language. In Extended Facile, agents are first-class functions that may be transferred to remote nodes for execution. The code for agent functions in Extended Facile can be transferred in a higher-level, platform-independent representation, as native-code executable instructions, or as a mixture of the two. The memory state of the function (i.e. its free variables and parameters) are marshalled and transferred by the system using a mechanism based on the underlying continuation-based compilation model of the language. The state of the agent is conveniently available in its continuation, and thus can be marshalled by the system in straightforward fashion. This continuation-based approach to capturing the state of native-code executable processes is clearly different from that of the stack-based approach of Process Introspection. In principle, both approaches are general, and could offer good performance. In practice, a limitation of the continuation-based approach is the relative scarcity of compilers that employ a continuation-passing model. Especially for performance-oriented languages such as C and Fortran, compilers generally employ the model that best matches the language and underlying processors, namely a stack based model. Compilers for functional languages such as Facile, ML, and Scheme more commonly use a continuation-based compilation model, but rarely offer performance comparable to lower level languages due to features such as first-class code and polymorphism. In practice, such languages are rarely used for high performance applications. Thus, given practical considerations, Process Introspection is a preferable approach in the context of high performance computing.

# Chapter 3

## Process Introspection

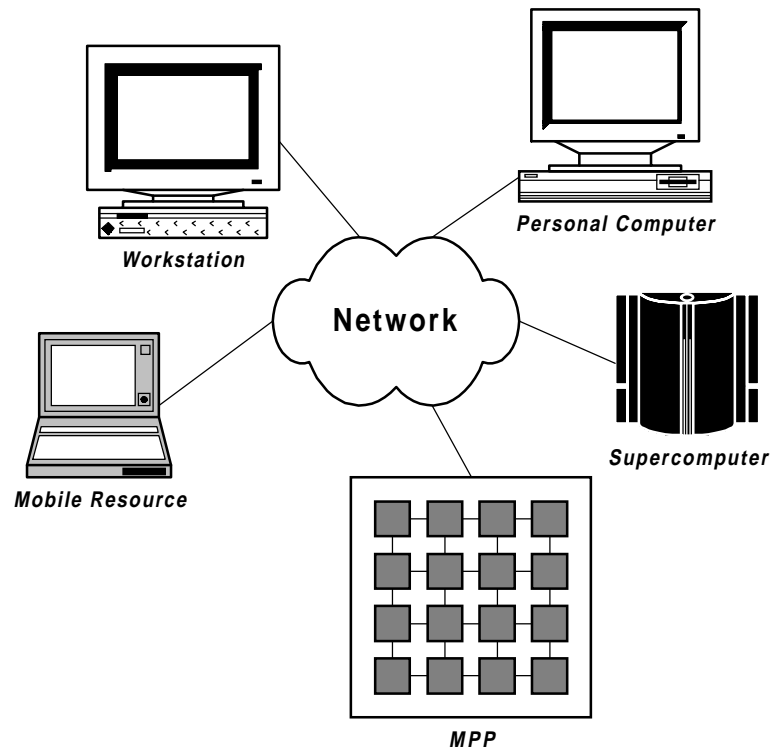
In this chapter, we describe in detail the Process Introspection solution to the heterogeneous process state capture and recovery problem. The key element of Process Introspection is the semantics-preserving modification of programs to incorporate state capture and recovery functionality, giving processes the ability to capture and recover their own state autonomously. This is an approach that is fundamentally different from existing solutions to the process state capture problem, which typically are based on an external agent (e.g. the operating system) examining and capturing the state of a process.

Before describing Process Introspection, we first discuss the system model (in Section 3.1) and the process model (in Section 3.2) on which Process Introspection is based. In Section 3.3 we describe in detail the capture of a process's internal state using Process Introspection. Finally, in Section 3.4 we address the issue of capturing a process's external state—the state of its interactions with other processes and its environment.

### 3.1 System Model

The system of interest is a heterogeneous network computing system, or *metasystem*. The system consists of a set of *nodes* connected by a set of *networks*, as depicted in Figure 3.1. Nodes in the system consist of at least a CPU and some local memory, and possibly other resources such as stable storage devices. For example, nodes might be personal computers, workstations, or MPP nodes. The networks connecting the nodes could be any medium that supports message-passing based cooperation. For example, the nodes might be connected by LANs, WANS, MPP interconnects, or shared memory. The nodes of the system might span a variety of architecture types and capabilities, and may run different base operating systems. We assume that the complete system is

programmed through, and managed by, an intermediate layer of software—the *metasystem software*. This software is responsible for providing the communication and task control mechanisms, and may provide other services, such as task placement support, a uniform file system, and so on.



**Figure 3.1:** A metasystem.

Applications that run in the system are programmed in a high-level language<sup>1</sup> and are decomposed into *tasks*, parts of the program that will execute independently, possibly concurrently, and possibly on different nodes in the system. The high-level language defining each task is transformed into a procedural intermediate representation of that task, which in turn is compiled to executable forms for all types of nodes in the system on which instances of the task may run. The intermediate representation of a task defines a *task class*, instances of which can be created or located by the metasystem software. The compiled executables associated with a task class are

---

1. In practice, our implementation of Process Introspection (described in Chapter 5 and Chapter 6) does not strictly prohibit modules coded at a lower level, for example critical inner loops implemented in assembly language. The assumption of a unified programming model simplifies our description of the Process Introspection algorithms and facilitates our correctness discussion in Chapter 4.

functionally equivalent *task implementations*. Within a task class, implementations may vary in performance, resource requirements, etc., but are assumed to compute equally valid results given the same input<sup>1</sup>. When a request is made to the metasystem software to create a task of a given class, the metasystem software selects a node on which to execute that task, selects the appropriate implementation of the task class for that node, and starts a process (as defined by the node's underlying operating system) on the node using the selected implementation. Thus, a process as viewed by the system is an executing task implementation, the meaning of which is defined by its task class.

## 3.2 Process Model

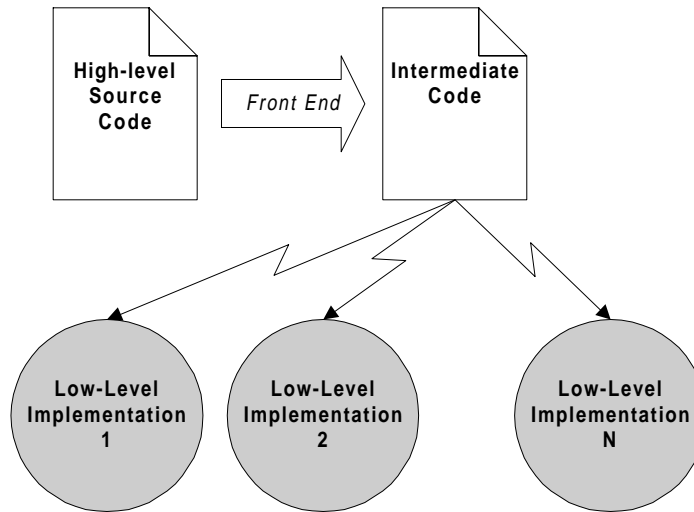
Before formally defining the Process Introspection mechanisms, we must develop a more precise definition of a process. Our basic process model is a standard, procedural, stack based model of computation. In this section we define this model more formally in order to provide a foundation for the description of the Process Introspection algorithms, and as a basis for the Process Introspection correctness discussion that will be covered in Chapter 4.

The fundamental meaning of a process is defined by its intermediate representation (IR) program (i.e. its task class), 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. A running process consists of a low-level program implementation (e.g. a binary executable containing object code for a given machine) and a set of dynamic program state elements that are manipulated by the program. As described in Section 3.1, many low-level implementations of the program might exist (e.g. for different computing platforms, with different levels and types of debugging information or optimizations), but exactly one version of the intermediate representation of the program exists

---

1. We do not specify that task implementations of a given task class produce identical outputs given the same input to accommodate issues such as varying floating-point representations and the possibility of non-deterministic tasks.

and defines the meaning of the process, as depicted in Figure 3.2.



**Figure 3.2:** The program model.

The intermediate representation of the program, which we will call  $P$ , consists of a set of subprograms,  $S$ , and a set of static state elements (i.e. global variables),  $G$ :

$$P = \{S, G\} = \{\{s_1, s_2, \dots, s_n\}, \{g_1, g_2, \dots, g_m\}\}$$

Each static state element,  $g_i$ , is a named, addressable, global data block that will be available throughout the lifetime of the program, and each subprogram,  $s_x$ , consists of a sequence of instructions,  $C_x$ , and a set of local variables,  $L_x$ —automatically allocated state elements available to an activation of the subprogram throughout its lifetime:

$$s_x = \{C_x, L_x\} = \{(c_i, c_{i+1}, \dots, c_j), \{l_{x,1}, l_{x,2}, \dots, l_{x,k}\}\}$$

The meaning of the program is defined in terms of its execution on an intermediate representation virtual machine (IR-VM). When the program is instantiated as a process, the IR-VM begins executing the program at the first instruction in a special subprogram,  $s_1$ . The IR-VM executes the instructions in  $s_1$ , continuing in sequence until the last instruction of  $s_1$  is executed or a subprogram termination instruction is encountered, at which point execution stops. The state of the IR-VM includes the IR program, a set of active state elements available to the running program, and



control flow state. The control flow state consists of a stack of subprogram activations, each of which records the identity of, the code location within, and the local state element activations associated with an individual subprogram activation.

The active state element set,  $A$ , consists of a set of statically allocated state elements, a set of dynamically allocated state elements, and a stack of automatically allocated data elements associated with the active subprograms. Statically allocated data elements are valid throughout the lifetime of the program. Dynamically allocated data elements can be created and deleted explicitly using special dynamic memory management instructions, and are active only after creation and before deletion. Automatically allocated data elements are the local state elements associated with a subprogram, and are automatically created and deleted on subprogram entry and exit. Note, state elements are defined at the level of the intermediate representation of the program, and may or may not map directly to the memory regions utilized by low-level implementations of the program.

The instructions contained in the programs fall into five categories:

- *State Modification* instructions alter the value of active state elements based on combinations of other state element values and/or constant values, without requiring any subprograms to be executed. For example, an instruction of this form might set one state element equal to the sum of three other state elements. State modification instructions are required to operate exclusively on active state elements—i.e. static state elements, dynamically allocated state elements that have been created but not yet deleted, and automatically allocated state elements associated with a subprogram that has been activated but has not yet terminated.
- *Branching* instructions alter the control flow of the program based on a boolean combination of active state elements and constant values. Based on the test, control flow can be directed to any other instruction in the same active subprogram.
- *Subprogram Activation and Deactivation* instructions activate and deactivate a given subpro-

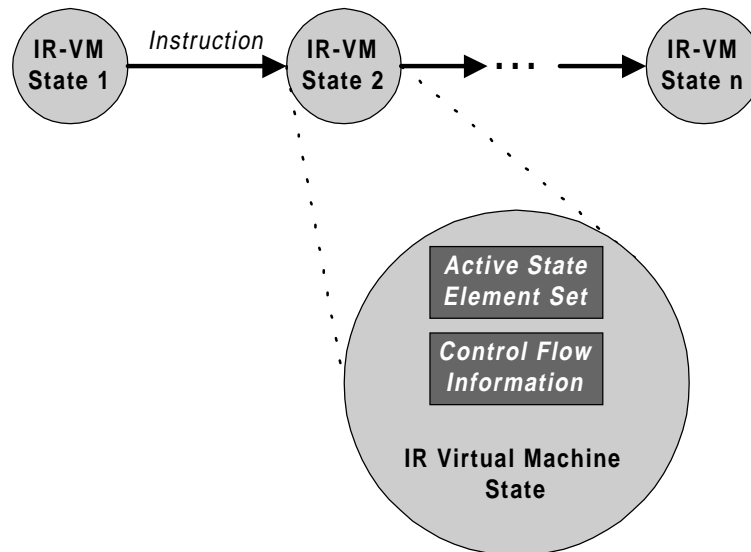
gram. On activation, the automatic variables for the specified subprogram are instantiated on an activation stack and are added to the active state element set, the return code location in the calling subprogram call is saved, and control flow shifts to the first instruction in newly activated subprogram (a conventional stack based execution model). Subprogram deactivation instructions terminate the currently active subprogram and return control flow to the call site from which the subprogram was activated. All automatically allocated local variables associated with the deactivating subprogram are deallocated.

- *Dynamic Memory Management* instructions create and delete dynamically allocated state elements. A dynamic allocation instruction creates a new state element with a given size and adds it to the active state element set. The state element will be valid until it is discarded by a dynamic deletion instruction.
- *Input/Output* instructions read or write the value of a state element from or to the program's environment, respectively.

This set of instruction kinds is general enough to allow our intermediate representation to serve as a target for any high-level programming language. The execution of each instruction by the IR-VM alters the IR-VM state in one atomic action, as depicted in Figure 3.3. Note, we have not specified any syntax or the precise semantics of our instructions—instead, we have defined the general attributes that the intermediate representation must exhibit. This level of specification is sufficient to describe Process Introspection and to discuss its correctness without unnecessarily limiting the possible implementations of the 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 and with different levels and types of debugging information or optimizations (see Figure 3.2), but each low level implementation must emulate the execution of the IR-VM, providing equally valid output given the same input. Low-level

implementations are expected to be primarily native-code executables that might be optimized to take advantage of certain properties that are not externally observable in the intermediate program—the intermediate representation defines the meaning of the program, but does not dictate the actual memory layout 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.

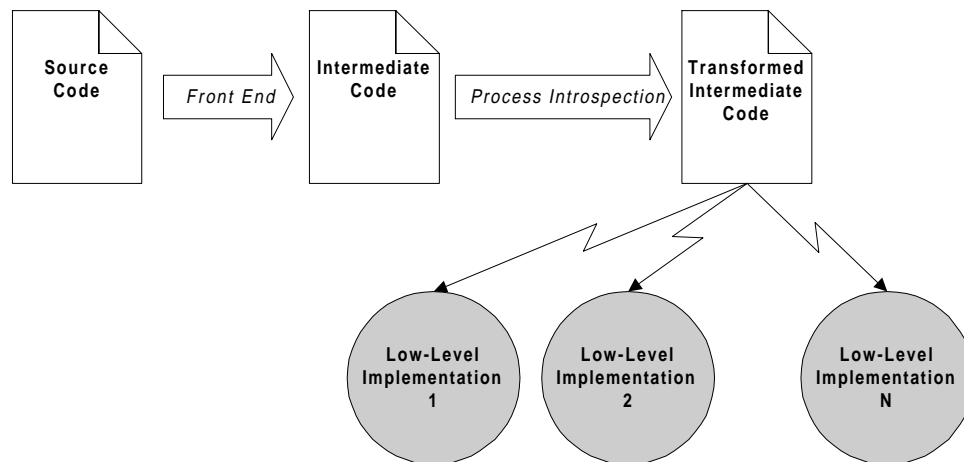


**Figure 3.3:** Program execution on the IR virtual machine.

### 3.3 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). This ability requires two specific modifications to the intermediate representation of the program: State capture transformations, and state recovery transformations.



**Figure 3.4:** Creation of an introspective program.

### 3.3.1 State Capture Transformations

The following four transformations are applied to the program in order to allow it to capture its internal state at special points in execution called poll points:

- A static state element (i.e. global variable) is added called *CheckpointStatus*. This variable is used by the program to indicate if a checkpoint has been requested or is in the process of being constructed. *CheckpointStatus* is initialized to indicate that no checkpoint has been requested, and that no checkpoint is in progress. The act of requesting a checkpoint from the process is thus equivalent to setting the value of *CheckpointStatus*. This might be accomplished within the process, providing a user-directed style of checkpointing [21], or might occur via an interrupt from the process's environment, or via shared memory with the process's environment. The later two forms permit a standard, externally requested process state capture.
- A new subroutine called *s<sub>chkpt</sub>* is added to the program. *s<sub>chkpt</sub>* contains instructions to output the values of all global variables and all active dynamically-allocated state ele-

ments.

- For  $s_{chkpt}$  to be able to output all of the dynamically-allocated state elements, the program must maintain a table that reflects 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 an up-to-date list of dynamically allocated state elements.
- 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 initial call to  $s_1$  for the program, a call to  $s_{chkpt}$  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 practice, 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*.

The above modifications to the program give it the ability to periodically poll for checkpoint requests. If a checkpoint is requested, the program outputs the values of the complete set of state elements along with the information necessary to reconstruct the program's control flow state (i.e. the subprogram activation stack and the current instruction location). Note, we have assumed that

the state of the program can be output in a universally readable format—i.e. for any state element in the process, we can describe its complete range of potential values in a format readable on any other type of computer system. In theory, this issue is not problematic, but in practice data representations differ among computer platforms, program variables can contain memory addresses (i.e. pointers), and thus storage of program state in a universally interpretable format is a significant design issue. We will defer discussion of this topic until Chapter 5, where we describe the implementation of a set of software mechanisms that support platform-independent capture and recovery of memory regions containing arbitrary data types.

### 3.3.2 State Recovery Transformations

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

- The range of values possible for the *CheckpointStatus* variable is expanded so that it can be used to indicate whether a restart is in progress or not. *CheckpointStatus* is initialized by input from the program's environment at the beginning of the base subroutine for the program to determine if a restart has been requested. In the event that a restart is requested at this point, the process can assume that its checkpoint is available to be read from a well-known location (e.g. from a given file or over a network connection).
- A new subprogram called  $s_{restart}$  is added to the program.  $s_{restart}$  contains instructions to read the values of all static state elements from the checkpoint and re-instantiate (i.e. re-allocate) the set of active dynamically-allocated state elements stored in the checkpoint.
- A prologue of instructions is added to each subprogram. This prologue 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 sub-routine from the checkpoint. If the subprogram is the initial activation of  $s_1$ , an instruction to call  $s_{restart}$  is also executed. Finally, the logical control flow location in the current subprogram activation is read from the checkpoint, and a branch to the associated code location is performed. This location may itself be another subprogram activation instruction, in which case the next subprogram in the stack is called and restores 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* is 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, again assuming that the state elements and control flow information can be stored in a universally recognizable format that masks architectural issues such as data representation (cf. Sun XDR [79]).

Of course, this claim is far from obvious, especially given that low-level implementations of the program may be optimized arbitrarily (with the constraint that they preserve the meaning of the program). 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, in Chapter 4 we 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 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 state capture and recovery mechanism as implemented in the intermediate code results in dependencies at poll points that cause certain optimizations across poll-points to be prohibited. We will examine the issue of correctness in detail in Chapter 4.

### 3.3.3 Optimizations

A side-effect of the described checkpoint mechanism is the deallocation of all memory on the call stack during the state capture, 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 stack frame to the checkpoint, the program should also save these values in 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. global variables 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 knowledge is available that all possible call chains resulting from a given subroutine call would contain no poll points, the mandatory poll point following that 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.

### 3.4 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. While producing a checkpoint, a process may be the destination of any number of messages that are in transit in the network system. The process may own locks to certain resources such as databases or hardware devices. The *external state* of the process can be large and complex to capture in a checkpoint. Nonetheless, any process state capture and recovery 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 the operating system or metasytem software) to provide some system support for checkpointing with respect to external state. For example, the MIST [18] system provides system support for checkpointing sets of processes communicating via the PVM interface, and the CoCheck system [84] provides similar functionality for MPI programs. These systems provide the ability to capture network state external to individual processes, allowing the process state capture mechanism to focus on internal state.

**Case 2, Wrapper Modules**—In some 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 (which is typically stored in kernel memory), but if it used wrapper routines to access all file operations, the

wrapper library could maintain an accurate internal record 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 time of the checkpoint). Using wrapper routines to capture a process's external state is a technique that has been demonstrated to be effective in projects such as Condor [54] and other such load-sharing tools used in homogeneous systems.

# Chapter 4

## Correctness Discussion

The Process Introspection mechanism modifies programs to allow them to checkpoint and restart at certain well defined poll points during execution. A program is given the ability to describe in complete detail certain intermediate states of its computation, and to restart from those states. Since a primary goal of this mechanism is to support restarts across different architecture and operating system platforms, and in particular to do so for optimized, native binary code executable programs, the correctness of the mechanism must be examined carefully. Given some forms of code optimization on different platforms, the intermediate states of a computation on one machine may not be consistent with the intermediate states of the computation on a different machine. It would seem possible that a checkpoint taken on one platform might not contain enough information to reconstruct an equivalent program state on a different platform.

In this chapter, we examine the correctness of the Process Introspection mechanism as developed in Chapter 3. In Section 4.1, we first discuss what correctness means in the context of the heterogeneous process state capture and recovery problem. In Section 4.2, we apply the developed definition of correctness to the Process Introspection mechanism, and discuss the underlying reasons why technique is correct. Finally, in Section 4.3, we provide illustrative examples of the correctness of Process Introspection in the presence of certain code optimizations. It should be noted that the discussion to follow does not constitute a formal proof of correctness, but rather a definition of state capture and recovery correctness, and a discussion of the features of Process Introspection that allow it to meet this definition. A fully formal proof of correctness is beyond the scope of this work.

## 4.1 Heterogeneous Process State Capture Correctness

### 4.1.1 General Definition

Before examining the correctness of the Process Introspection mechanism, we must first define what it means for any heterogeneous process state capture and recovery mechanism to be correct. Intuitively, a heterogeneous (or homogeneous) process state capture mechanism can be considered correct if it does not affect the behavior of a program. Thus, to define process state capture and recovery correctness precisely, we first define program correctness.

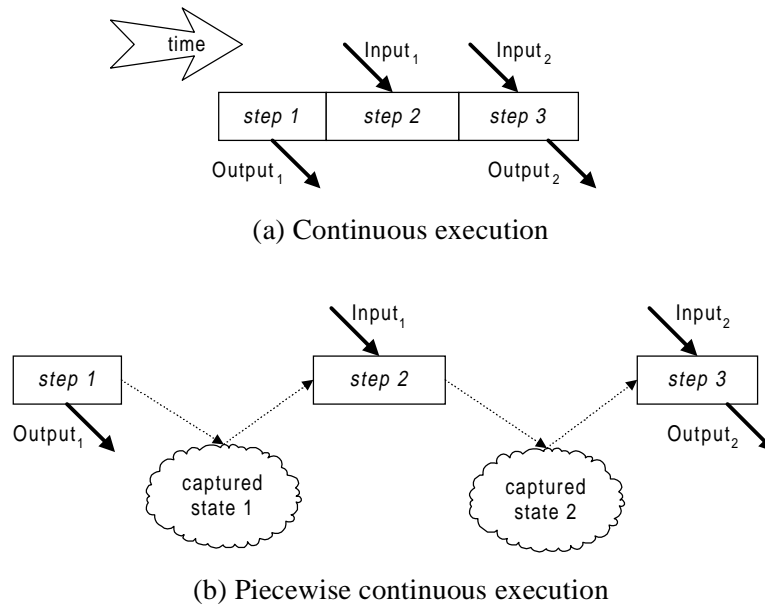
**Program Correctness**—Our definition of program correctness is based on the principles of axiomatic semantics. We assume that a program is specified in the form of a set of preconditions, the IR code for the program, and a set of postconditions. Preconditions and postconditions are expressed in the form of logical assertions about IR program state, input, and output. We further assume that axioms and inference rules are provided for all of the IR language instruction types, so that given a postcondition and an IR instruction, we can determine the weakest precondition for the instruction. We define a program execution to be correct if starting from the postcondition, final program state, and output, we can work backwards through the IR program using the rules of axiomatic semantics and eventually derive the precondition. For our purposes, a key attribute of this definition of program correctness is that two program executions (for example, as might be performed by two different low-level implementations of the IR program) may produce different outputs and yet still both be considered correct. For example, consider a program that produces a solution vector as output. Suppose that the postconditions for the program specify that the magnitude of this vector must be less than some constant  $\epsilon$ . Given the same input, different executions of the program might produce slightly different result vectors. However, if both output vectors have the required property of magnitude less than  $\epsilon$ , these different executions would both be correct under our definition despite the difference in their output. If two executions of a program given the same input both meet our definition of correctness, we define these executions to produce *equally*

*valid* (although possibly not identical) output.

Given this definition of program correctness, we define the correctness of a process state capture and recovery mechanism in terms of its ability to preserve program correctness. That is, when presented with the same input, a program should produce equally valid output irrespective of the number of times it is checkpointed and restarted during the course of execution. Of course, this does not imply anything about the correctness of the program being checkpointed and restarted—incorrect programs will be just as incorrect after being checkpointed and restarted. The correctness of a state capture and recovery mechanism is fundamentally rooted in its preservation of a program's semantics, whatever they may be.

To more precisely specify the correctness of a state capture and recovery mechanism, we introduce the concept of a piecewise continuous execution of a program. Consider the execution of a program as a sequence of steps during which some sequence of inputs is accepted and some sequence of outputs is produced, as in Figure 4.1(a). An execution of the program without interruption due to state capture or recovery can be considered a *continuous execution* of the program. Alternatively, a complete execution of the program interrupted by state captures and recoveries, but where no part of the program is repeated or omitted, and identical inputs are provided at the appropriate points in execution, can be called a *piecewise continuous execution* of the program, as in Figure 4.1(b). For any state capture and recovery mechanism, we require that any piecewise continuous execution of a program produce equally valid outputs given the same inputs as a legal continuous execution of the program.

The correctness of a *heterogeneous* state capture and recovery mechanism is based on the ability of any of a number of different implementations of a program to be restarted from a checkpoint created from any of the other given implementations while preserving the validity of output produced by the program when given the same input. Note, we do not require a heterogeneous piecewise continuous execution to produce the exact same output given the same input as any sin-



**Figure 4.1:** Equivalent continuous executions of a program.

gle-platform continuous execution. Issues such as varying rounding and discretization error might result in a piecewise continuous execution producing outputs different from any continuous execution on a single platform with the same input, although the outputs might be equally valid given the specification of the program. Based on the notion of correctness for a heterogeneous process state capture and recovery mechanism, and on the idea of piecewise continuous execution, we state a general correctness condition for heterogeneous state capture and recovery mechanisms:

**General Correctness Condition**—Given a task class and a set of task implementations associated with that task class, any piecewise continuous execution of the task with any assignment of task implementations to the execution segments must produce output that is equally valid to a legal continuous execution of the task given the same input (although the output need not be the same as that produced by a continuous execution of any single task implementation).

#### 4.1.2 Restricted Definition

The correctness condition given in Section 4.1.1 is very general. In fact, we can state a more

restrictive correctness criterion that will be easier to demonstrate for the Process Introspection mechanism, and that is fully sufficient. This correctness criterion will be based on the restriction that low level implementations of a program contain certain points of consistency with the intermediate representation of the program, and that checkpoints and restarts occur only at these points of consistency. Although technically not all heterogeneous process state capture and recovery mechanisms need to observe this restriction, the observance of this restriction makes correctness more readily demonstrable.

Assume that we restrict the state capture mechanism to operate only at certain points during the execution of any implementation. Assume also that we restrict these points to correspond to well defined points in the execution of the intermediate code representation of the program if it were run on an IR-VM. More carefully stated, the checkpoint mechanism will be restricted to operate at certain points during the execution of a low-level implementation (i.e. a native code binary) where the state of this running implementation can be used to fully describe a state of the intermediate representation program executing on the IR-VM (an IR-VM state). Specifically, the state of the implementation at these points will be used by the state capture mechanism to identify those state elements that would be active in the IR-VM and to determine their values. Furthermore, the implementation's state will be used to determine the point in control flow of the IR-VM—i.e. the current active subroutine, the next instruction to be executed, and the subprogram activation stack. The resulting checkpoint will then contain a complete description of the state of the intermediate representation of the program executing on the IR-VM.

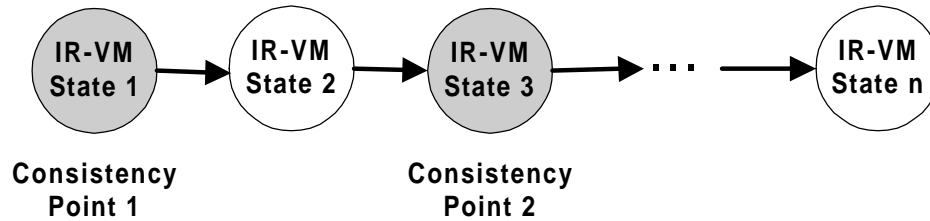
As a natural consequence of this restriction on state captures, we also restrict state recovery to occur at these same well defined consistency points. In other words, a checkpoint (i.e. a full description of the state of the intermediate representation program) will need to be used by the state recovery mechanism to re-initialize a low-level implementation to return it to a consistency point. This will require that these execution states of any low-level implementation be fully restor-

able based only on the information contained within an IR-VM state. For example, if some temporary storage not corresponding to any state element in the IR program were used by the low-level implementation (e.g. a common subexpression computed before the consistency point were reached), the state recovery mechanism would be responsible for restoring this storage based exclusively on the IR-VM state description.

Based on the above simplifying restrictions on the operation of a heterogeneous process state capture and recovery mechanism, we can develop more specific criteria for correctness that will be possible to demonstrate for the Process Introspection mechanism. We now outline these criteria:

**Criterion 1—Checkpoint Correspondence.** We define an IR-VM state of an executing IR program to be the set of active state elements and their values along with a control flow state consisting of a subprogram activation stack defining the code location in each active subprogram. The execution of each IR instruction by the IR-VM transitions the running program to a new IR-VM state. Thus, an execution of the intermediate program can be described as a sequence of IR-VM states,  $E = (IS_1, IS_2, \dots, IS_n)$ . We designate a certain possibly non-contiguous subsequence of these IR-VM states as the sequence of *consistency IR-VM states* for the program, as in Figure 4.2. The consistency IR-VM states will be the points during execution at which the state capture and recovery mechanism will be permitted to operate. We say that a low-level implementation of the program *reaches* a given consistency IR-VM state  $IS_k$  if its execution up to a given point is indistinguishable from the execution of the intermediate code program up to that IR-VM state, and if its state at that point can be used by the state capture mechanism to fully describe  $IS_k$ . Criterion 1, the checkpoint correspondence criterion, requires that every low level implementation of the program *reach* each IR-VM state in the sequence defined by the execution of the intermediate code program on the IR-VM. This criterion simply requires that each low level implementation have the same set of points of consistency with the intermediate code program in the same order. In other words, each one will potentially be the basis of the exact same set of checkpoints.





**Figure 4.2:** Consistency points.

**Criterion 2—Restart Correctness.** The second criterion ensures the ability to recover state correctly. Assume that we are given a checkpoint created by any low-level implementation at a given consistency IR-VM state,  $IS_k$ . Criterion 2 requires that if the state recovery mechanism uses this checkpoint to restart any low-level implementation of the program, and if the program is then presented with the remainder of its input required after the checkpoint was taken, the program will correctly reach  $IS_{k+1}$ . If demonstrated to hold for all  $k$ , this criterion is sufficiently powerful to imply that the program will run correctly to completion after being restarted. Once the program reaches  $IS_{k+1}$  from  $IS_k$ , it is indistinguishable from a program restarted directly at  $IS_{k+1}$ , and therefore will proceed correctly on to  $IS_{k+2}$ , and so on. By induction, the program will run to completion correctly from a restart if criterion 2 is satisfied.

If a state capture and recovery mechanism meets the above two criteria, then it will naturally satisfy the general correctness condition stated in Section 4.1.1. Due to the restriction that state capture and recovery must occur only at consistency points, elements of any piecewise execution of a program will consist of periods of execution bounded by two consistency points in the program. Thus, if the above the two correctness criteria hold, program meaning will be preserved across consistency points, and any piecewise continuous execution of the program will produce valid outputs.

It is important to note an important underlying assumption upon which the above restricted correctness criteria are based. In requiring that all low-level executables reach the same consis-

tency points in the same order, we have assumed that program control flow is both deterministic and platform independent. In a nondeterministic program, one execution might pass through any number states never encountered by another execution of the program. This would preclude the requirement that all task implementations reach the same consistency points. Furthermore, if control flow is platform-dependent, for example if it is sensitive differences in rounding error, precision, discretization error, etc., versions of the program on different platforms might proceed through arbitrarily different states. For programs that exhibit either nondeterminism or platform-dependent control flow, the criteria presented in Section 4.1.2 are too restrictive. Thus, in discussing the correctness of Process Introspection, we will first assume that programs are both deterministic and exhibit control flow that is not dependent on differences in the available platforms (e.g. data representation differences). We later address the issue of these somewhat restrictive assumptions separately.

## 4.2 Process Introspection Correctness

We can demonstrate the correctness of the Process Introspection mechanism by showing that introspective programs satisfy the correctness criteria described in section 4.1.2. First, we must define the set of consistency IR-VM states in the transformed intermediate representation of an introspective program. Next, we must argue that criterion 1 is satisfied—i.e. that all implementations of the transformed intermediate representation pass through the same consistency IR-VM states in the same order that the intermediate code would. Finally, we must argue that criterion 2 is satisfied—i.e. that any implementation of the transformed intermediate representation when started from the checkpoint for a given consistency state  $IS_k$  will correctly reach state  $IS_{k+1}$ .

Defining the consistency IR-VM states for the transformed intermediate code is trivial—these are simply the states of the intermediate code just before each poll point introduced into the code is reached. Recall, at each poll point in the transformed code (i.e. when any IR-VM state is reached

whose next instruction is a poll point), the program examines the value of a variable to determine if a checkpoint should be written. Furthermore, the IR-VM states just before each poll point is executed are the only states during execution for which the process will produce a checkpoint. Being the only points in the code at which checkpoints can be written, IR-VM states just before each poll point are thus also the states that restarts are limited to recover. Given these attributes of the IR-VM states at poll points, they naturally satisfy the definition of consistency IR-VM states.

Next, we must demonstrate that any valid low-level implementation of the transformed intermediate code will reach each consistency IR-VM state in the order prescribed by the execution sequence the intermediate code would follow if executed on a virtual machine as described in Chapter 3 (i.e. criterion 1). In fact, the observance of criterion 1 is a fundamental requirement of any correct low-level implementation of the intermediate code because we have introduced new potential inputs and outputs for the intermediate representation of the program.

Consider the first poll point encountered by a transformed program. At this poll point, the transformed intermediate program specifies that the value of a variable set by some input from the environment be examined by the program. The implementation can make no assumption about what the value of this variable will be—it may be cleared, or may have been set to request a checkpoint. If the value indicates a checkpoint request, the program as transformed is defined to output the values of all state elements as they would be valued in the intermediate program. To correctly emulate the IR-VM execution of the program, any low-level implementation must be prepared to accept this synchronous checkpoint request, and must produce a process state description as output (containing all of the state elements active in the intermediate program) exactly as the intermediate program would have. We can be certain that all correct low-level implementations of the intermediate program will reach  $IS_1$  and produce an identical checkpoint describing all of the intermediate program's state elements if so directed, precisely because this is part of the input/output specification of the transformed IR program. After  $IS_1$  is passed, the intermediate code specifies that when

$IS_2$  is reached, a similar process is observed, and again any correct implementation must be prepared to accept input and produce the complete values of state elements as output if directed. We can argue similarly for  $IS_{k+1}$  having reached  $IS_k$ , and thus by induction, we know that any correct implementation will reach each consistency IR-VM state in the appropriate order. Furthermore, given the nature of the state capture output statements added to the code at each poll point, we have the added required attribute of the checkpoint mechanism that it correctly obtain a complete consistent description of  $IS_k$  when it captures that state.

Given that all implementations correctly reach all consistency IR-VM states in the specified order, we must further show that if a low level implementation is restarted to some consistency IR-VM state  $IS_k$ , that it will correctly reach  $IS_{k+1}$  (criterion 2). Again, the observance of this criterion follows as a result of the low level implementation's requirement to produce the same output given the same input that the intermediate representation would have produced if executed on the IR-VM. Recall, at the beginning of execution of a transformed introspective program, the program accepts input to determine if a restart should be accepted. When a restart is requested, the program sets a special variable to indicate that a restart is in progress, reads in the values of all active state elements in the intermediate program and restores the state of control flow. In other words, all of the information in  $IS_k$  is read by the program and integrated into the program's state. Now, consider the state of the intermediate code executing on an intermediate representation virtual machine at  $IS_k$ . On one hand, the program can reach this state through normal (i.e. non-restart) flow of execution. On the other hand, the intermediate program can reach this state by performing a restart—i.e. accepting input from the environment. In the intermediate code, these (identical) states are both reachable given certain inputs from the environment. However, both have the same effect—the program continues normal execution to  $IS_{k+1}$  and beyond. Since this is the specified behavior of the intermediate code, it must be respected by any low level implementation. That is, any low level implementation of the transformed intermediate code must be able to accept input of

the state  $IS_k$ , restore to the state  $IS_k$ , and proceed to state  $IS_{k+1}$ , since that is the specified input/output behavior of the transformed intermediate code.

### 4.2.1 Restrictions

The above discussion of the reasons why programs transformed to support Process Introspection observe the correctness criteria presented in Section 4.1.2 was based on the assumptions of deterministic execution and platform independent control flow. In practice, either or both of these restrictions may fail to hold for programs of interest. Thus, we now address the ability of Process Introspection to correctly handle programs that violate these assumptions.

The issue of nondeterminism can be addressed by extensions to the above model and discussion. Assuming that we extend the model of computation described in Section 3.2 to incorporate nondeterminism, a process would consist of a set of possible IR-VM state progressions instead of a single sequence of IR-VM states. In implementing the nondeterministic IR program, each low-level executable would need to allow the emulation of any of these state progressions. Thus, we could again mark certain states (in the multiple possible state progressions) as consistency states, and require the restricted correctness criteria of Section 4.1.2 with the modification that they hold for any arbitrary legal IR-VM state progression specified by the program. As in the deterministic case, Process Introspection transformed programs will observe the correctness criteria due to introduced input and output dependencies. For a given possible IR-VM state progression (i.e. for a given set of outcomes of the nondeterministic events in the program), the transformed programs will encounter the same poll points in the same order, and will have the ability to recover program state at each poll point.

The second important assumption on which the correctness discussion of this chapter is based is platform-independent control flow. For example, we have assumed that low-level program implementations do not exhibit different control flow behavior due to varying data formats (e.g.

floating-point representation). This may at first appear to be a significant limitation. For example, given different floating-point representations, the same iterative algorithm might converge in a different number of iterations. Given different numbers of bits in the representation of an integer, a counter might overflow in one low-level executable version of a program, but not in another. In general, program control flow may vary arbitrarily depending on data representation and other architectural features. However, given the discussion in this chapter, platform-independent control flow is a fundamental requirement for correct state capture and recovery through Process Introspection. In practice, this constrains the operation and applicability of the state capture and recovery mechanism. For example, if an environment contains some nodes that support 64-bit integers and others that support 32-bit integers, programs that wish to migrate between nodes arbitrarily must restrict the range of integer values they use to the 32-bit range. Alternatively, programs could use the full 64-bit integer range but would then need to be restricted to execution only on 64-bit platforms.

Given the fundamental requirement of platform-independent control flow, and the target application domain of high-performance scientific computing, an immediate concern is control-flow dependence on floating-point representation and algorithms. While integer representation only leads to the issue of different representable ranges, floating-point representation and algorithms lead to basic issues such as range, precision, and rounding error, and related issues such as discretization error, catastrophic cancellation, and convergence error. In general, there are many ways in which floating-point representation can affect program control flow, and it is often difficult to analyze the precise dependency of control flow on floating-point representation issues. Fortunately, differences in floating-point representation are becoming rare in practice due to the widespread adoption of the ANSI/IEEE Standard 754-1985. This standard dictates both floating-point representation and how basic algorithms such as rounding should be performed. Even given identical representations, programs can still contain platform dependencies due to floating-point arith-

metic, for example due to order of expression evaluation. However, floating-point standardization significantly reduces the number of ways in which programs can exhibit platform-dependent control flow.

### 4.3 Correctness Examples

Whereas the above discussion describes the underlying reasons why Process Introspection is correct, it is illustrative to consider the effects of Process Introspection on certain code optimizations, and the reasons that Process Introspection operates correctly in their presence. In this section, we examine three such examples.

#### 4.3.1 Loop Reordering

In our model, the code transformations to support state capture and recovery are applied to IR programs before any low-level executable implementations are produced. Since code optimizations are applied after the state capture and recovery transformations, the application of some back-end optimizations may be hindered by Process Introspection. A simple example of this effect is loop reordering. Consider the code fragment that performs an **axpy** operation depicted in Figure 4.3(a). On certain machine architectures, executing the loop in reverse order as in Figure 4.3(b) might improve performance.

<pre>axpy(float a, x[N], y[N]) {     int i;      for(i=0; i&lt;N; i++)         y[i] = a*x[i] + y[i]; }</pre>	<pre>axpy(float a, x[N], y[N]) {     int i;      for(i=(N-1); i&gt;=0; i--)         y[i] = a*x[i] + y[i]; }</pre>
(a) Original	(b) Reordered

**Figure 4.3:** Axy loop.

Although this transformation of the code is completely legal (i.e. it produces the same results with

no observable difference except performance), such an optimization would be disabled in the presence of the Process Introspection transformations if a poll point were placed within the loop. Consider the transformed pseudocode in Figure 4.4.

First, notice that in this transformed code, during any iteration of the loop the value of **CheckpointStatus** can indicate that a checkpoint has been requested. If a checkpoint has been requested, the code produces as output the values of the variables **i**, **a**, **x**, and **y**. For a low-level implementation to meet this added requirement of the code, the implementation must be able to produce the values of **i**, **a**, **x**, and **y** during any iteration, and as specified by the loop ordering of Figure 4.4. The likely result of this added requirement is that the loop of Figure 4.4 will not be transformed to use the ordering of Figure 4.3(b).

---

```

axy(float a, x[N], y[N]) {
    int i;

    if(CheckptStatus==Restart) {
        input values of i, a, x, and y
        goto poll_point1;
    }

    for(i=0; i<N; i++) {
        y[i] = a*x[i] + y[i];
    poll_point1:
        if(CheckptStatus==Checkpoint) {
            output values of i, a, x, and y
            return;
        }
    }
}

```

---

**Figure 4.4:** Axy loop, transformed.

Another result of the transformations on the code is that alternate paths into the loop have been added. The prologue added to the code fragment determines if a restart has been requested. If it has, the values of **i**, **a**, **x**, and **y** are read from input, and the flow of execution branches directly



to the poll point in the loop. If the loop were reordered as in Figure 4.3(b), the effect of this jump into the loop would not produce the desired effect—the elements from 0 to  $i$  would have the `axpy` operation applied instead of the elements from  $i$  to  $N-1$  as specified by the code in Figure 4.4. Again, to preserve the meaning of the program, the loop reordering optimization would likely be disabled.

It should be noted that this discussion does not imply that loop reordering is always disabled by Process Introspection transformations. In this example, had the poll points been placed outside the loop (i.e. before and/or after, but not in the loop body), the loop could have been reordered as in Figure 4.3(b). This leads us to the fundamental observation that the policy for placing optional poll points used in Process Introspection transformations can have serious impact on the optimizations that will be applicable to the transformed code. The above discussion could just as easily have been applied to many other common code optimizations such as loop unrolling, vectorization, and so on.

### 4.3.2 Inline Subroutines

In Section 4.3.1 we discussed why Process Introspection transformations can inhibit the application of certain code optimizations. In this section, we consider an example of an optimization that is not affected by Process Introspection transformations, and that does not affect the correctness of Process Introspection: function inlining. Assume that the example function of Figure 4.3(a) is instrumented as in Figure 4.4. Furthermore, assume that the function is invoked at some point in a program, causing the placement of a mandatory poll point, as in Figure 4.5.

---

```

poll_point2:
    axpy(A, X, Y);
    if(CheckptStatus==CheckptInProgress) {
        output A, X, Y, etc.
        return;
    }

```

---

**Figure 4.5:** Axy invocation, transformed.

In one low-level implementation of the program fragment depicted in Figure 4.5, the invocation of the **axpy** function might compile to an actual function invocation. In this case, the actual call stack (or at least this part of it) would match that of the IR virtual machine. In a different implementation, the invocation of **axpy** might be inlined to improve performance—i.e., in place of the invocation to **axpy** in Figure 4.5, the function body of Figure 4.4 would be inserted (with the appropriate variable substitutions). In this case, during the **axpy** invocation, the call stack of the implementation would not match that of the IR-VM. Despite this invocation stack mismatch, the correctness criteria are still observed. Recall, the correctness criteria did not require that control flow/invocation stack information match the IR-VM state at consistency points, only that a correct description of the IR-VM state could be generated based on the implementation.

Consider the operation of both implementations (inlined and non-inlined) if a checkpoint is requested when the program is executing the **axpy** loop and reaches **poll\_point1**. In the non-inlined case, the local variables are saved, the **axpy** function returns, the caller saves its local variables, and so on. In the inlined case, the output is identical—only the internal control flow is different. When the poll point is encountered, the “local” variables for the **axpy** routine will be output, and the function immediately returns. The return in this case will translate into a jump to the statement immediately following the **axpy** invocation. Since this statement is another poll point, the remaining local variables of the caller are output, and the function returns. The state descriptions produced in each case are identical.

Furthermore, each implementation can recover equivalent states based on the resulting checkpoint. In each case, the stack is recovered up to the point at which the invoking function containing the code in Figure 4.5 is restored. When this function’s stack information is recovered, its local variables are restored, and the flow of execution branches to the code location **poll\_point2**. In the non-inlined case, the **axpy** routine is invoked, and it in turn recovers its state and continues execution. In the inlined case, the inlined function recovers its “local” state and jumps into the

**axpy** loop. In both cases, the exact same checkpoint data is input, the appropriate state is recovered, and the process resumes execution correctly.

This leads to the general observation that, whereas the Process Introspection transformations seriously affect the application of some optimizations, others such as function inlining and constant hoisting are unaffected by Process Introspection.

### 4.3.3 Code Motion

The examples discussed thus far have been relatively straightforward. In the case of loop reordering, certain poll point placements could almost certainly inhibit the application of the optimization. In the case of function inlining, the optimization was unaffected by the Process Introspection transformations. In many cases, the interaction between optimizations and the Process Introspection transformation is more complicated. We now consider an example of an optimization that is still desirable and applicable after the Process Introspection transformations, but the application of which is complicated: code motion. Consider the code fragment in Figure 4.6(a). This

---

```
example() {
    int i, a, b, c;

    for(i=0; i<(a+b+c); i++) {
        loop body
    }
}
```

---

(a) Original code

---

```
example() {
    int i, a, b, c;
    int temp;

    temp = a+b+c;
    for(i=0; i<temp; i++) {
        loop body
    }
}
```

---

(b) Code motion applied

**Figure 4.6:** Code motion.

loop represents an opportunity for loop invariant code motion. Assuming that the loop body does not alter **a**, **b**, or **c**, the transformed code fragment presented in Figure 4.6(b) could result in improved performance, saving the redundant evaluation of the expression (**a+b+c**) on each itera-

tion of the loop.

Now consider the transformation of the code in Figure 4.6(a) to support Process Introspection, as in Figure 4.7. If we naively apply the code motion optimization depicted in Figure 4.6(b) to this transformed version of the code, the resulting program would be incorrect. If the function were called during normal execution, the value of `temp` would be computed as expected and the loop would execute correctly. If, however, the function were entered during a state recovery operation, the initialization of `temp` would be skipped, and the loop would have an undefined termination condition.

This discussion does not imply that code motion could not still be correctly applied to the Process Introspection transformed code, but the application of this optimization would need to take into account the new path into the loop block. This could be handled by an additional initialization of `temp` in the restart prologue, as depicted in Figure 4.8. Although many optimizers are

---

```

example() {
    int i, a, b, c;

    if(CheckptStatus==Restart) {
        input i, a, b, and c
        goto poll_point;
    }

    for(i=0; i<(a+b+c); i++) {
        loop body

        poll_point:
        if(CheckptStatus==Checkpoint) {
            output values of i, a, b, and c
            return;
        }
    }
}

```

---

**Figure 4.7:** Transformed code motion example.

probably not sophisticated enough to perform this extra step in order to take advantage of the code

motion optimization in this case, all back end compilers must preserve the meaning of the transformed version of the program as depicted in Figure 4.7, and thus the correctness of Process Introspection is assured.

---

```

example() {
    int i, a, b, c;
    int temp;

    if(CheckptStatus==Restart) {
        input i, a, b, and c
        temp = a+b+c;
        goto poll_point;
    }

    temp = a+b+c;
    for(i=0; i<(a+b+c); i++) {
        loop body
        poll_point:
            ...
    }
}

```

Additional initialization of `temp` →

---

**Figure 4.8:** Transformed example, code motion applied.

Our examples of the interactions between the Process Introspection transformations and back-end compiler optimizations have focussed on the issue of correctness. We have observed that the meaning of the Process Introspection transformations is specified fully at the common IR level of program representation before optimizations have been applied, and thus the meaning of these transformations must be preserved by back-end compilers. The practical manifestation of this rule is that certain optimizations will not be performed by back-end compilers because of the Process Introspection transformations. This leads to an important practical observation: the application of Process Introspection, especially the placement of poll points, must be performed with care to avoid serious performance impact. In Chapter 7, we discuss an empirical study of the effects of Process Introspection on optimized programs. In practice, we have found that it is possible to apply Process Introspection effectively without losing the benefits of optimizing compilers.

# Chapter 5

## Library Implementation

In preceding chapters we have described Process Introspection on an abstract level, leaving many detailed design and implementation issues open. In this chapter and in Chapter 6, we describe a full working implementation of a Process Introspection system. This chapter describes a library interface and implementation that supports the application of Process Introspection transformations to programs written the C programming language [46]. This run-time library can be employed by a programmer in order to apply the Process Introspection transformations by hand, or can serve as a target interface for a compiler that applies these transformations automatically. The implementation of such a compiler is described in Chapter 6.

### 5.1 System Implementation Overview

We have constructed a prototype implementation of a Process Introspection system consisting of a run-time support library, the Process Introspection Library (PIL), which provides an interface for writing program modules supporting state capture and recovery, 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 runs on a variety of workstation and PC platforms, including Sun workstations running Solaris or SunOS 4.x, SGI workstations running IRIX 5.x and 6.x, IBM RS/6000 workstations running AIX, DEC Alpha workstations running Digital Unix or Linux, and PC compatibles running Linux or Microsoft Windows 95/ NT.

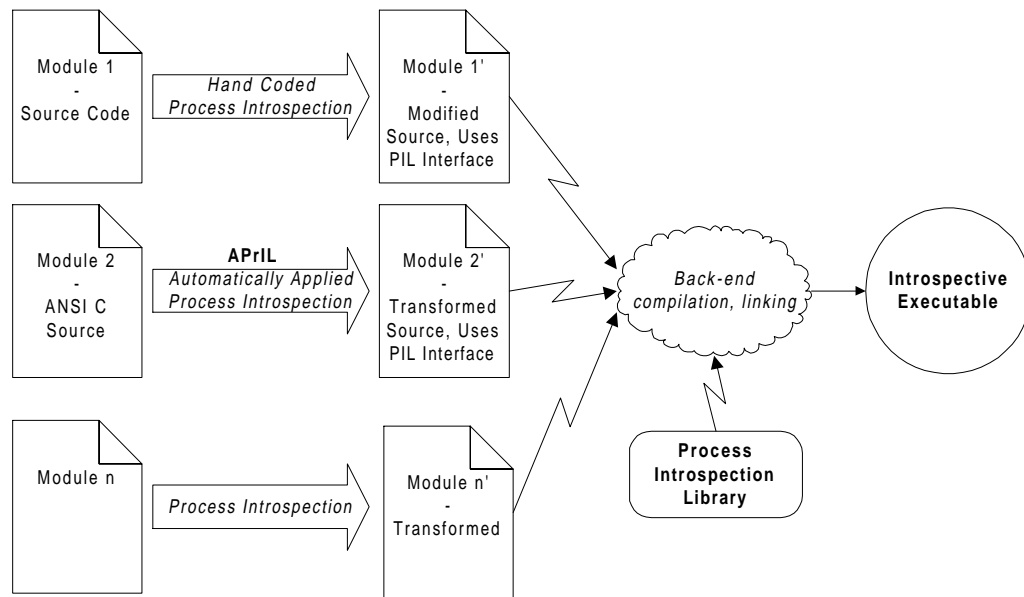
Usage of the system generally takes one of two forms. In the case of modules that have architecture dependencies (for example, modules that are wrappers around external services such as the file system), the modifications to support Process Introspection must be hand coded (as in the case

of Module 1 in Figure 5.1). 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 polling for checkpoint requests periodically), and to provide a platform-independent mechanism for saving the data associated with the module. For example, in coding a file interface module with state capture and recovery capabilities, 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 to 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 introspective 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 transformed to incorporate state capture and recovery mechanisms by applying the APrIL source code translator, which also uses the PIL as a run-time interface. This usage mode, which corresponds to the case of Module 2 in Figure 5.1, requires no extra work on the part of the programmer to apply Process Introspection.

In this chapter, we discuss the implementation of the Process Introspection Library; the implementation of the APrIL compiler is covered in Chapter 6. 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 state capture and recovery 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 coordi-

nating the activities of modules at checkpoint and restart time.



**Figure 5.1:** Using the Process Introspection system.

## 5.2 Library Overview

In applying the Process Introspection transformations, we have thus far assumed that a mechanism is available for saving and recovering the values of data regions in a process's address space in a platform-independent format. In practice, such a mechanism is not readily available. The Process Introspection Library, or PIL, provides mechanisms to capture and recover general ANSI C data structures in a platform-independent manner, allowing the Process Introspection transformations to be more easily applied to programs coded in C. The library comprises the following modules:

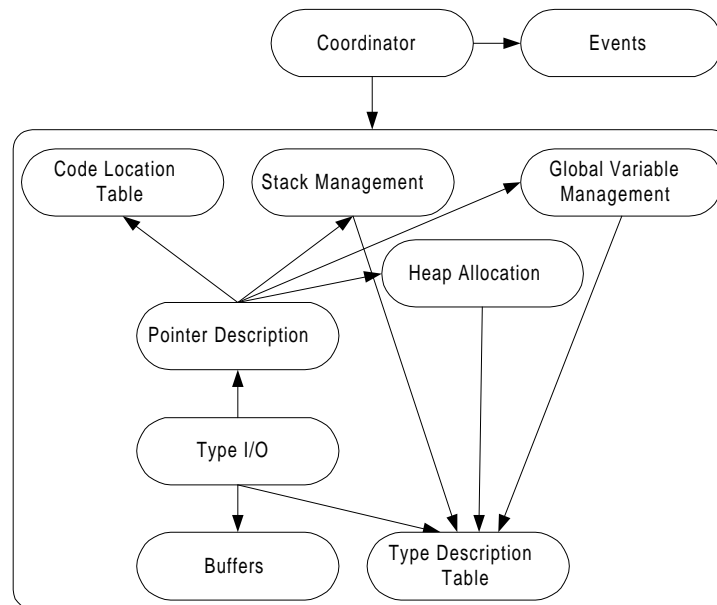
- *Buffer Module*—The buffer module abstracts the checkpoint storage medium from the program (e.g. whether the checkpoint is to be written to a disk file, over a network, etc.), and provides platform-independent I/O for the basic types defined by the C language.
- *Type Description Table*—This table supports the registration of type descriptions, and provides information (such as size and alignment values) for registered types. The table has a set of pre-



- defined basic types, and supports composition of types into vectors and records.
- *Typed I/O*—This module permits general typed I/O of data regions in the process's address space. Using a registered type table description of the data region, and the PIL buffer mechanism, the typed I/O routines can read or write any data region in the process's address space.
  - *Global Variable Management*—This module supports the registration of global variables at process startup time. These globals will be captured automatically at checkpoint time and recovered at restart time.
  - *Heap Allocation*—Heap allocation wrapper routines allow the library to keep track of the types and sizes of all dynamically allocated data structures in the process. These data structures are saved automatically at checkpoint time and are recovered at restart time.
  - *Pointer Description*—This module generates platform-independent, logical descriptions of pointers in order to support the typed I/O module (which may be required to read and write data structures containing memory addresses).
  - *Stack Management*—This module provides routines for reading and writing stack allocated data structures from and to the checkpoint. Routines are also provided to register the addresses of certain local variables in order to support the pointer description module.
  - *Code Location Table*—This table allows the registration of function pointers in order to support the operation of the pointer description module.
  - *Events*—The events module provides a mechanism for the combination of separate transformed, introspective modules. This module allows handler functions to be registered by modules to be called at key points in execution, such as at process startup, on checkpoint, and on restart.
  - *Coordinator*—The coordinator provides a generic interface for requesting checkpoints and restarts. In practice, the coordinator is a replaceable module that determines how the rest of the library interacts with the process's environment. For example, the coordinator encapsulates the

external checkpoint request mechanism and the checkpoint location mechanism.

The PIL's modular structure and inter-module dependencies are depicted in Figure 5.2. We now describe the PIL module interfaces and implementation details in greater depth.



**Figure 5.2:** PIL modules and dependencies.

## 5.3 Library Implementation

### 5.3.1 Buffer Module

The PIL buffer module provides the basic storage abstraction for the Process Introspection Library. This module encapsulates the medium used to store and retrieve a process's checkpoint data, which may reside on stable storage, may be retrieved from over a network, may be in memory, and so on. Buffer implementations provide operations for writing and reading raw sequences of bytes to and from the abstracted storage. Based on these low-level operations, storage routines for platform-independent input and output of higher-level data types can be layered. The current library provides implementations for on-disk buffers and in-memory buffers.

Layered on the basic buffer interface, the PIL provides routines for reading and writing typed

data from and to a buffer in an architecture-independent format. This interface is responsible for masking differences in byte ordering, floating-point representation, size, and so on. When a PIL buffer is created at checkpoint time, it is tagged automatically to indicate the supported PIL data format that the process will use to store data (i.e. the native data format for the architecture on which the process is running). Later, when the buffer is used to restart the process, the data it contains is converted from the stored data format to the restarting processor's data format automatically as it is read by the restarting process, a protocol known as *receiver-makes-right* [96].

Given this receiver-makes-right 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 (such as Sun XDR [79]), 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 in light of the very small number of data formats actually in use 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. Furthermore, the size and complexity of the library can be reduced by a simple conditional compilation optimization: a version of the library compiled for a given platform need only include the routines to translate between other data formats and the native data format, reducing the actual number of routines in any one version of the library to  $O(n)$ .

An example of PIL buffer usage is depicted in Figure 5.3. It is worth noting that separate routines are provided for reading and writing scalar and vector values. The scalar interface supports a pass-by-value interface, allowing scalar variables to be written to the checkpoint without inhibiting the allocation of those variables to registers by back-end compilers.

---

```

PIL_Buffer *buf;
float x[N];

buf = PIL_FileBufferCreate("Checkpoint.data");
PIL_BufferPutInt(buf, N);
PIL_BufferPutFloats(buf, x, N);

```

---

**Figure 5.3:** PIL buffer usage example.

### 5.3.2 Type Description Table

The PIL buffer interface provides mechanisms for reading and writing memory regions containing scalars or vectors of the basic C data types. In general, memory regions may contain user-defined structured data types. To checkpoint or restore such memory regions, the PIL must have a description of the regions' data storage layout. The PIL provides an interface to a table that maps integer type identifiers to logical type descriptions, and that allows the creation of new type descriptions based on the composition of existing ones. The PIL type description table is not unlike a type 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 identifiers for the basic types supported by ANSI C, and supports an interface for describing vectors and records based on existing types. In order for the PIL to capture or recover a memory region, the memory layout of that region should be capable of being described as a basic data type supported by the type table, a linear vector of some number of elements of a type described by an entry in the type table, or a structure containing a list of elements, each of which is described by an entry in the type table.

When transforming a program to utilize the PIL for Process Introspection, all required types must be registered by the program on process startup. An example of two basic type registrations using the type table interface is provided in Figure 5.4. In this example, the type descriptor `t1` is initialized to describe a type consisting of an array of 100 double precision floating-point values.

The second type descriptor, **t2**, is initialized to describe a structure type. The structure type is then defined to include two elements: an integer and an element of type **t1** (an array of 100 doubles).

---

```

int t1, t2;

t1 = PIL_RegisterArrayType(PIL_Double, 100);
t2 = PIL_RegisterStructureType();
PIL_AddStructureField(t2, PIL_Int);
PIL_AddStructureField(t2, t1);

```

---

**Figure 5.4:** Type registration examples.

The internal implementations of the type registration routines generate and save internal type descriptions. For arrays, the type table index of the array elements and the number of array elements are recorded. For structures, a list of elements is generated, with each list node containing the type table index and the offset of the field. In addition to this basic descriptive information, the generated type table entry includes size and alignment information for the type. This information is generated for all data formats supported by the library (based on the basic data type sizes and alignment rules for those formats) in order to support the receiver-makes-right protocol for user-defined data types.

### 5.3.3 Typed Input/Output Module

Based on the mechanisms for platform-independent I/O of basic data types provided by the PIL buffer module, and on the ability to describe user defined types provided by the type description table, the PIL typed I/O module supports platform-independent input and output of memory regions containing user defined data types. As depicted in Figure 5.5, usage of the typed I/O interface is straightforward. The routines take parameters indicating the PIL buffer to read or write, the type table index of the data elements contained in the memory region being captured or recovered, a pointer to the memory region, and a count of the data items contiguously stored in the region.

The internal operation of these functions can proceed in one of two modes: automatic or user-

defined marshalling. In automatic mode, the memory region is stored or retrieved based on the type description found in the type table. For example, in the case of a structure, each field listed for the structure in the type table is input or output using a recursive application of the appropriate typed I/O function, the recursion reaching a base case in the event that the parameter type table index corresponds to a basic data type.

---

```
void PIL_WriteTypedArray(PIL_Buffer *buf, int type_table_index,
                        void *ptr, int count);
void PIL_ReadTypedArray(PIL_Buffer *buf, int type_table_index,
                       void *ptr, int count);
```

---

**Figure 5.5:** Typed I/O interface.

User-defined marshalling mode is an alternative to automatic mode that requires additional programming effort on the part of the library user, but can offer better performance. To employ user-defined marshalling mode, the library user writes input and output functions for a desired data type, and registers these functions with the PIL, specifying the type table index with which they should be associated (see Figure 5.6). The user defined marshalling functions export an interface

---

```
struct user_struct { (structure contents) };
                . . .
user_struct_index = PIL_RegisterStructureType();
                . . .
void input_user_struct(PIL_Buffer *, void *, int);
void output_user_struct(PIL_Buffer *, void *, int);

PIL_RegisterMarshallingFunctions(user_struct_index,
                                input_user_struct, output_user_struct);
```

---

**Figure 5.6:** User defined marshalling function registration.

similar to the generic typed I/O routines, but do not require a type table index since they are assumed to operate on a single, known type. User defined marshalling functions can improve per-

formance by avoiding the need to interpret type descriptions at run-time to perform marshalling. Furthermore, in coding the user defined marshalling functions, a user can take advantage of semantic information not available in the type table. For example, a large matrix might offer better run-time performance if stored using a standard dense-matrix format while the process is active, but might be more efficiently checkpointed and recovered using a sparse matrix format.

If user defined marshalling functions are defined for a data type, the typed I/O routines defer to these; otherwise, automatic mode is employed. In either case, the typed I/O module allows any memory region whose contents are described in the type description table to be stored in and recovered from PIL buffers.

### 5.3.4 Global Variable Table

The PIL provides an automated mechanism for saving and restoring the values of global variables at checkpoint and restart times, respectively. This 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 at process startup time. An example of global variable registration is depicted in Figure 5.7. In this example, an integer scalar, **N**, and a vector of 100 double precision floats, **X**, are registered with the global variable table. Note, the registered memory

---

```
PIL_RegisterGlobal(&N, PIL_Int, 1);
PIL_RegisterGlobal(X, PIL_Double, 100);
```

---

**Figure 5.7:** Global variable registration.

regions are also permitted to contain user defined types, and thus global variable registration would generally follow type table initialization at process startup time.

Each global variable registered in the PIL is assigned a unique identification number. When the process performs a state capture using the Process Introspection mechanism, the data contained

in each registered global memory region is saved along with the region's associated identification number using the typed I/O interface. When the process is restarted and must recover its state, the global variable registration is repeated, and the values of all global regions are recovered, again using the typed I/O interface. The restarting process may execute on a different platform, in which case the actual addresses of the registered global regions may have changed, but this issue is masked by the use of the unique identification numbers to match checkpointed data with the appropriate global memory region.

### 5.3.5 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 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. Allocation routines corresponding semantically to the standard C library routines `malloc`, `calloc` (which clears the allocated memory region), and `realloc` (which re-sizes an allocated memory region) are provided by the library, in addition to a `free` wrapper routine.

An example of PIL heap wrapper usage is depicted in Figure 5.8. Note that the PIL allocation routines require a parameter specifying the type being allocated, allowing the library to later correctly store and retrieve the allocated memory region using the typed I/O routines. It is also worth noting that the routines allocate a region sized based on number of data items rather than on number of bytes (as with C `malloc`). The size of the returned block is computed by the routines based on the type table entry for the allocated type.

As with global variables, dynamically allocated memory blocks are assigned unique identification numbers by the PIL heap management routines. When the process state is captured, the data



---

```

float *x;
struct user_struct *s; /* Registered in type table as */
                        /* "user_struct_index" */

x = (float *)PIL_Malloc(PIL_Float, 100);
s = (struct user_struct *)PIL_Calloc(user_struct_index,1);

PIL_Free(x);

```

---

**Figure 5.8:** PIL heap management example.

contained in each heap memory block is saved along with its identification number and a memory block description consisting of a type table index and number of elements corresponding to the memory layout of the region. When the process is restarted, each heap block is re-allocated based on the saved memory block description (if necessary—in the case that the process has not stopped or migrated, the memory block may still be available), and its contained data is recovered from the checkpoint. As in the case of global variables, the addresses of dynamically allocated memory regions may change after a state capture and recovery of the process. In Section 5.3.6 we examine how memory relocation is automatically masked from user code by the PIL.

### 5.3.6 Pointer Description

Thus far, we have only considered the case of memory regions containing simple numerical data types that can be transformed between different data formats in a straightforward manner, and have ignored the issue of pointers. Memory addresses (i.e. pointers) contained within memory blocks are inherently platform-dependent—i.e., even if the appropriate data format conversion is performed to allow the process on a different platform to interpret the same address value, the address will almost certainly have different meaning in the new address space. Furthermore, the operation of the PIL at state recovery time can move the location of memory regions (e.g. heap allocated blocks), and thus a mechanism for masking this from user code is required.

When captured in a checkpoint, pointers 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. To address these requirements, the PIL supports a pointer description mechanism based on the assignment of unique identification numbers to every memory block of interest in the program. A logical pointer description is a tuple containing a memory block type, a memory block identification number, and an offset into the memory block. The pointer analysis module provides an interface for generating logical descriptions of memory locations and for resolving these logical descriptions into actual memory addresses, as depicted in Figure 5.9.

---

```
PIL_PoInterDescription *PIL_DescribePointer(void *addr);

void PIL_ResolvePointer(PIL_PoInterDescription *d, void **ptr);
```

---

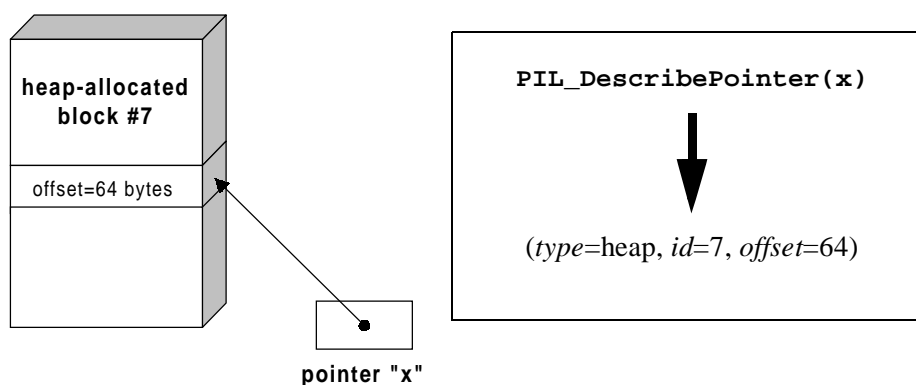
**Figure 5.9:** Pointer description interface.

The implementation of pointer description is based on simple case analysis; a pointer is allowed to 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 (i.e. a function pointer)
- A special value that has meaning in the program (such as NULL in C)

To determine the appropriate case, the pointer description routine searches for the address it must describe in the global variable table, in the heap allocation table, in a local variable table (to be described in Section 5.3.7) and in the code location table (to be described in Section 5.3.8). Based on the table in which the memory block to which the pointer refers is found, and the pointer's offset into that memory block, a pointer description is generated, as in Figure 5.10. We note an impor-

tant optimization to the search process described here; since it is typical for the different types of memory regions (i.e. global, stack, heap, code text) to reside in different regions of a process's address space, the search process described above can be pruned significantly by maintaining upper and lower memory address boundaries for the memory regions registered in each table. For example, the global variable table keeps track of the beginning of the registered memory region with the lowest starting address, and the end of the registered memory region with the highest starting address. Thus, when a search is performed during a pointer description operation, it can quickly be determined if the pointer can possibly refer to a registered global memory region.



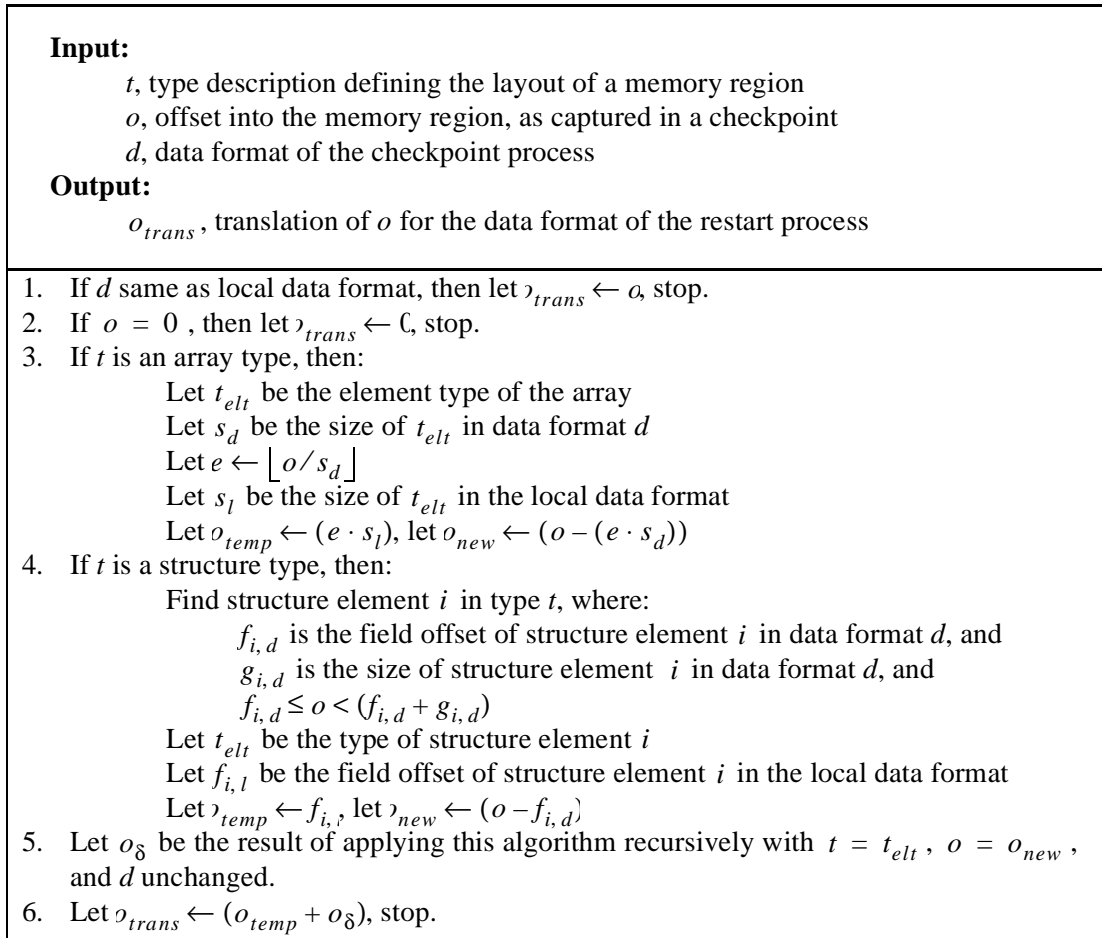
**Figure 5.10:** Pointer description.

Pointer resolution, the mapping of logical pointer descriptions to actual memory addresses, is required when process state is recovered. Given a pointer description, the pointer resolution operation looks up the appropriate memory block in the indicated table. For example, if the pointer description block generated in Figure 5.10 were resolved, memory block 7 would be found in the heap allocation table. The only remaining step, given the appropriate base address for the referred-to memory block is the addition of the pointer offset. Of course, since the process may be restarted on a platform with a different data format, the offset may need to be transformed. Again, this protocol is a receiver-makes-right strategy, which improves performance when the restart data format is compatible with the checkpoint data format—in this case, we may directly add the offset and return the resulting address.

### 5.3.6.1 Pointer Offset Translation

Pointer offset translation is based on the layout of the memory region into which the pointer points. The fundamental assumption of the translation algorithm is that a pointer offset into a memory region when added to the base address of that region should point to the beginning of a data element whose type is registered in the type description table. For example, an address might point to the beginning of an array of structures, or might point to the beginning of one of the structures in the array, or might point to the beginning of a field inside one of the structures in the array, and so on. Given this assumption, the algorithm operates by finding the largest data structure inside the memory region in question, the beginning of which the offset would have pointed to in the checkpoint process's address space. Once this structure is found, its offset into the corresponding memory region in the restart process's address is computed, which is the resulting translated offset.

The details of the pointer offset translation algorithm are presented in Figure 5.11. Steps 1 and 2 of the algorithm cover the base cases where the checkpoint and restart data formats match, or the offset is zero, respectively. In either of these cases, the offset need not be transformed. Step 3 is performed if the highest level memory layout of the region into which the pointer refers is an array. In this case, the algorithm determines the element of the array into which the offsets refers, sets  $o_{temp}$  be the local offset of that array element, and lets  $o_{new}$  be the checkpoint process's offset into that array element. Similarly, step 4 is performed if the highest level memory layout of the region is a structure type. In this case, the algorithm determines the structure field into which the offset points, sets  $o_{temp}$  to the local data format offset of that field, and sets  $o_{new}$  to the checkpoint process's offset into that structure field. After either step 3 or 4 is performed, the algorithm is applied recursively to transform the remaining offset,  $o_{new}$ , into the appropriate array element or structure field. This transformed offset remainder,  $o_{\delta}$ , is then added to  $o_{temp}$  to determine the final transformed offset.



**Figure 5.11:** Pointer offset translation algorithm.

### 5.3.6.2 Pointer Resolution Ordering

In our discussion of pointer resolution thus far, we have assumed that the memory referred to by a logical pointer was allocated and resident in memory. During a process restart, this assumption will not hold if a pointer resolution is attempted before the referred-to memory region (e.g. a heap allocated block) is reallocated. This problem is an artifact of the fact that the PIL does not perform all memory reallocation before memory reinitialization (i.e. reading the data from the checkpoint into memory) is started. Instead, as the PIL reallocates each memory block, it attempts to reinitialize that block. In the case of numerical data this is safe, but for pointers, this aggressive reinitialization strategy can lead to resolution failures. To address this issue, the PIL maintains a

list of unresolved pointers during a restart. Each element on the list contains a description of a memory region containing an unresolved pointer, and the logical pointer description needed to later resolve it. The PIL provides a routine that iterates over the unresolved pointer list, resolving those unresolved pointers for which the referred-to memory region was subsequently reallocated. This operation is invoked automatically at the end of a restart, but may be invoked safely at any time.

### 5.3.7 Stack Management

Since pointers may refer to stack allocated memory regions, the addresses and types of certain local variables need to be registered with the PIL to support pointer description. An example that uses the PIL interface to register stack allocated memory blocks is depicted in Figure 5.11. Not all

---

```

int n;
double x[100];

PIL_RegisterLocal(&n, PIL_Int, 1);
PIL_RegisterLocal(x, PIL_Double, 100);
    . . .
PIL_UnRegisterLocal(&n);
PIL_UnRegisterLocal(x);

```

---

**Figure 5.11:** Local variable registration example.

local variables need be registered with the PIL, only those that might be referred to by pointers (e.g. those whose addresses are used in an expression). The registration of local variables is performed only to support pointer description, not to provide an automatic save and restore mechanism, as with global and heap allocated blocks. As described in the Process Introspection model (Chapter 3), local variables are saved explicitly at poll points encountered after a checkpoint request, and are restored in subroutine prologues during a restart.

To support the explicit checkpoint and recovery of local variables, the PIL provides an inter-

face layered on the PIL buffer mechanism and the typed I/O module. During a checkpoint, each subroutine on the stack initializes an in-memory PIL buffer into which it writes its local state, thus supporting the restart optimization described in Section 3.3.3. This process is depicted in Figure 5.12(a). Similarly, during a restart, each call frame advances to the next element in a list of in-memory stack frame PIL buffers and then reads its local state, as depicted in Figure 5.12(b). The transfer of in-memory PIL buffers containing stack data to and from the actual checkpoint buffer is performed automatically by the PIL.

---

```

if (PIL_CheckptStatus==PIL_StatusCheckpointNow) {
    PIL_InitStackFrame(804); /* Specify required size */
    PIL_SaveStackInt(n);
    PIL_SaveStackDoubles(x, 100);
        . . .
}

```

---

(a) Saving local variables at a poll point

---

```

if (PIL_CheckptStatus==PIL_StatusRestartNow) {
    PIL_NextStackFrame(); /* Move to next buffer */
    PIL_RestoreStackInt(n);
    PIL_RestoreStackDoubles(x, 100);
        . . .
}

```

---

(b) Restoring local variables in a subroutine prologue

**Figure 5.12:** Explicit stack save/restore interface examples.

### 5.3.8 Code Location Table

Since pointers can refer to function entry points, the PIL must maintain a table that maps function pointers to logical identifiers to support pointer description. 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 PIL code location table interface, as depicted in Figure 5.13.

---

```
int (*f)();

PIL_RegisterFunctionPointer(rand);

f = rand; /* Had to register rand so the pointer */
          /* f can be described by the PIL      */
```

---

**Figure 5.13:** Function pointer registration.

### 5.3.9 Event Module

The event module provides the primary mechanism for separate modules to interoperate with respect to state capture and recovery, and for modules to customize their checkpoint and restart behavior. The event module allows a program module to register function callbacks that will be invoked automatically by the system on process startup, process checkpoint, and process restart. The most common use of the event mechanism is the declaration of a process startup event handler by a module that must register type descriptions, global variables, and so on.

To understand the importance of this mechanism for customizing a module's state capture and recovery behavior, consider the case of a file interface wrapper module. Besides the normal activities of saving and restoring the data in memory blocks (as is done by every module, and which is typically automated using the PIL), the file module must perform extra actions. 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 that would be executed in response to checkpoint and restart events.

### 5.3.10 Checkpoint Coordinator

Thus far, we have not discussed the issue of how checkpoints or restarts are requested from a process, how the checkpoint data is located and wrapped in a PIL buffer, and how the automatic



checkpoint and restart services provided by the PIL are performed (e.g. global variable recovery, heap reallocation and recovery, etc.). These issues are encapsulated by a replaceable checkpoint coordinator module. The coordinator might initiate checkpoints internally based on a scheduling algorithm, might accept checkpoint requests via messages received over a network, and so on. To allow the PIL to adapt to different environments, the operation of the coordinator is left intentionally loosely defined.

Although it is a replaceable module, the coordinator must interact with the rest of the library and with user code in a well defined manner. This interaction is defined by the coordinator interface:

- To initiate a checkpoint, the coordinator sets the value of a special global variable, **PIL\_ChckptStatus** to the value **PIL\_StatusCheckpointNow**. Similarly, to request a restart, the coordinator sets the value of **PIL\_ChckptStatus** to **PIL\_StatusRestartNow** at process startup.
- When the user code completes a checkpoint (using the Process Introspection mechanism described in Chapter 3), it should invoke the coordinator function **PIL\_DoneCheckpoint()**, which in turn will automatically save the registered global and heap allocated memory regions, and transfer the in-memory stack frame buffers to the checkpoint.
- Before initiating a restart, the coordinator automatically recovers the state of all global and heap allocated memory regions, and transfers the PIL buffers containing the user stack frames into an in-memory list.

Within this general framework, the coordinator's operation can be customized to a range of environments. The default Unix implementation coordinator included with the PIL accepts checkpoint requests via a signal, the handler for which sets the **PIL\_ChckptStatus** to indicate the checkpoint request. The checkpoint is stored in a file whose location is determined from a Unix environ-

ment variable. Restarts are initiated at process startup time based on the contents of the indicated checkpoint file. If the file indicated by the checkpoint environment variable is found to be non-empty, the restart is initiated automatically.

## 5.4 System Service Wrappers

In Section 5.3, we discussed the implementation of the core state capture and recovery mechanisms provided by the PIL. These mechanisms provide automation of the capture and recovery of internal process state, but do not address the issue of external state—the status of the process’s interactions with external system services. As described in Section 3.4, external state capture and recovery can be supported via system interface wrapper modules. As an example of system service wrappers to support state capture and recovery, we have implemented an auxiliary file interface library layered on the PIL.

The Process Introspection File (PI-File) library provides an interface corresponding to the C standard I/O interface, except that the identifiers in the PI-File interface are prefixed with the string “**PI\_**”. An example of the interface usage is depicted in Figure 5.14. A replacement

---

```

PI_File *f;

f = PI_fopen("data", "w+");
PI_fprintf(f, "%d, %f\n", i, x[i]);

```

---

**Figure 5.14:** Process Introspection file interface usage.

“**stdio.h**” header file is provided with the library to allow existing code implemented using the C standard I/O interface to use the PI-File interface automatically through a set of macro definitions.

The operation of the PI-File library provides automatic capture and recovery of a process’s open files, file pointers, and standard I/O buffers. The library does not capture and recover actual

file contents of open files, although this policy would be a straightforward modification to the PI-File implementation. The current PI-File implementation would be suitable for an environment where processes utilize state capture and recovery to support migration, but would not be safe for backwards error recovery in general, since the effects of file updates would not be rolled back automatically, and thus restarting processes could be presented with inconsistent file states. Despite these limitations, the PI-File library implementation is a good example of how system services can be wrapped in a module to support state capture and recovery.

The implementation of the PI-File library is layered on the services provided by the PIL, and on the C standard I/O interface. The PI-File library maintains a list of open file descriptions, which is updated by the `PI_fopen()` and `PI_fclose()` routines. Each open file description contains a pointer to an associated C standard I/O **FILE** structure, and a record of the file name and flags with which the file was open (e.g. read, write, append, etc.). When PI-File operations are performed on an open file, the associated C standard I/O operation is performed on the corresponding C standard I/O **FILE** structure.

The automatic checkpoint and recovery of the PI-File interface is based on two fundamental PIL features: heap allocation and events. First, the PI-File file records (i.e. **PI\_FILE** structures) used by the library are allocated using the PIL heap allocation routines. This provides automatic reallocation of the PI-File open file list, but more importantly, it supports the correct capture and restoration of **PI\_FILE** pointers in the user code. Recall, any pointers captured and recovered using the PIL interface must refer to a memory block registered in one of the PIL tables. Use of the PIL heap allocation routines for the PI-File open file list ensures that this rule is observed.

Reallocation of the open file list is only part of the problem of capturing and recovering the open file state. When the process is restarted, it must reopen any files that were open at checkpoint time and restore the process's file pointers. To accomplish this, the PI-File library uses the PIL event mechanism. The PI-File library registers event handlers to be called on checkpoint and on

restart. The checkpoint handler determines and saves the file pointer associated with each open file (using the standard I/O `ftell()` routine), and flushes the standard I/O buffers associated with any open files. The restart event handler iterates over the open file list, reopens the files with the appropriate flags, and restores file pointers (using the standard I/O `fseek()` routine). From the perspective of the user application, the result of this combination of event handler activities is the automatic capture and recovery of a process's open file state.

# Chapter 6

## The APrIL Source Code Translator

The programming interface provided by the PIL automates several aspects of Process Introspection, such as platform-independent I/O, but is still relatively low-level. At this level of automation, the programmer is still required to perform code modifications such as poll-point placement and prologue generation manually in order to employ Process Introspection. Fortunately, for platform-independent programs this process can be automated by a source code translator. Using the Sage++ toolkit [11], we have implemented this idea in the APrIL compiler. In this chapter, we examine the design of the APrIL compiler, and the fundamental specific transformations it employs.

### 6.1 Intermediate Representation

The model described in Chapter 3 required that the input to the translator be in a common intermediate representation. An intermediate representation for Process Introspection should balance the following goals:

- Platform-independence.
- High-level language independence.
- A base of reusable existing tool support (e.g. front-ends for various languages, back-ends for various platforms, optimizers, etc.).

For our prototype design of APrIL, the intermediate code we selected is ANSI C. APrIL transforms input ANSI C code, producing as output new ANSI C code that has been modified as described in Chapter 3, and that utilizes the PIL as a run-time interface. The resulting C code can then be compiled using any ANSI C compiler. C meets the first requirement, architecture-independence, if certain platform-dependent features (e.g. the “**asm**” directive) and inherently platform-

dependent programming-practices (e.g. relying on the representation of basic data types) are disallowed. Although 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. Optimizing back-end compilers are widely available. Front-end source-to-source translation tool-kits are also available; for example, the Sage++ library [11] (which was used to implement APrIL) provides an object-oriented interface to parsing, manipulating, and transforming C++ (and thus C) using a set of C++ object classes. It should be noted that the use of C 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 this chapter could be performed to implement the automatic checkpoint code, as will be discussed in greater depth in Chapter 8.

## 6.2 APrIL Transformations

In this section, we present the fundamental set of transformations performed by the APrIL compiler. Throughout this section, we make certain assumptions about the input code that simplify both the description and the implementation of the transformations in question. These assumptions are:

- All local variables are declared in the outermost scope of the function.
- All function calls appear only in simple C expression statements (e.g. not within the conditional of a loop or “**if**” statement, etc.).<sup>1</sup>
- At most one function call appears in any statement (e.g. an expression such as **f(x) + g(y)** is illegal).

---

1. We adopt the definition of expression statement as given by Kernighan and Ritchie in The C Programming Language [46] in the grammar for the C language presented in Appendix A. See the *expression-statement* production.

- In an expression statement containing a function call, the function call must be guaranteed to be evaluated before any other subexpressions with side-effects. For example, the actual parameters to all function calls (which are evaluated before the function is invoked) must be side-effect free expressions.
- No “**union**” data structures are declared.

For the remainder of Section 6.2 we will assume that the above restrictions hold for the input code. In fact, although the design and implementation of the transformations described in this section are based on these assumptions, the APrIL compiler as a whole performs a pre-processing step on general C input code to transform it to satisfy these assumptions (except the “**union**” restriction). Thus, the actual input code for APrIL need not observe these restrictions, which would be a severely limiting and unnecessary hindrance to the usability of the tool. We will examine the implementation of the APrIL pre-processor in Section 6.3.

### 6.2.1 Poll Points

One of the most fundamental transformations performed by APrIL is the insertion of poll points. In accordance with the model described in Chapter 3, APrIL inserts poll points throughout the code it transforms. Recall that poll points are the points at which the program checks for a checkpoint request, and if required captures the state of the current call frame. Using the PIL as an interface, polling for checkpoint requests simply involves examining the value of a global variable (**PIL\_ChkptStatus**) that is set to indicate that a checkpoint request has been received. Immediately following a 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 branches to a function epilogue that saves the actual parameters and local variables in the frame (described in greater detail in Section 6.2.3). 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, and are marked by APrIL using a C label inserted to allow a restarting process to jump to the poll point at which a given stack frame was checkpointed. Of course, since labels are not a first class data type in C, the checkpointing process must save some other data value to record the poll point location at which a stack frame checkpoint was generated. Using the PIL interface, this value is simply an integer that will map one-to-one to poll points within a given function. An example of an optional poll point is depicted in Figure 6.1.

---

```

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

```

---

**Figure 6.1:** An optional poll point.

Mandatory poll points expand on the design of optional poll points. In accordance with the Process Introspection model, APrIL inserts mandatory poll points after every function call statement in the code<sup>1</sup>—recall, these mandatory poll points are required to implement the stack capture operation based on the native function return mechanism as described in Chapter 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 as implemented by APrIL utilize 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

---

1. Recall that we have assumed that function calls appear in the input code only in simple C expression statements. In fact, function calls in C are just another type of expression, and can thus appear anywhere a valid expression can (e.g. a function call might be a parameter to another function call, which might be part of the conditional for an “**if**” statement, and so on). To perform the transformations as described in the model, APrIL extracts functions from complex expressions and statements, and reduces them to simple C expression statements containing a single function call. Details of this transformation are covered in Section 6.3.



checkpoint is initiated immediately following a normal function return). A bit in the `PIL_ChkptStatus` variable is set as soon as any poll point has been encountered after a checkpoint request, and thus the mandatory poll point code can simply test this bit's value to differentiate between the two cases. If the checkpoint is already in progress when the mandatory poll point is encountered (i.e. the checkpoint was begun in a higher call frame), the poll point marks the location immediately preceding the function call as the point at which the restarting process should resume in the function. This will later allow the stack to be recovered using the native function call mechanism. If the checkpoint is not in progress when the mandatory poll point is encountered, the checkpoint request must have been received after the last poll point in the previous function invocation was passed. Thus, the previous function call must have completed normally, and the restart code location selected is the statement immediately following the function call. An example of a mandatory poll point is depicted in Figure 6.2.

---

```

_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 6.2:** A mandatory poll point.

### 6.2.1.1 Poll Point Placement Policies

The placement of poll points in the code is a critical performance issue for APrIL. If poll points are placed so that they occur too frequently, the introduced overhead and impact on back-

end compiler optimizations 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 a large 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 should be selected. Alternatively, if checkpoint-request wait times must be very low (for example, if the checkpoint will be used for code migration to effect load sharing), then 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 based on the hypothesis that a set of relatively simple heuristic placement strategies can achieve a low degree of performance perturbation, and can at the same time provide low checkpoint request wait time on average. In Chapter 7, we present the results of a detailed performance study that substantiates this claim. Here, we present the details of the APRIL poll point placement heuristics.

The goal of the poll point placement policy is to select some set of statements in the input program, immediately following which poll points will be generated. In principle, every executable statement in the input program is a valid candidate—a trivial policy to implement could select every statement as a poll point, but would certainly result in high run-time overhead. The APRIL poll point placement heuristics narrow the selection process by only considering locations in the input program that are the last statement in the body of a loop<sup>1</sup>. This pruning of the possible search space is based on two observations: First, it is unlikely that a single basic block of code will execute for a significant portion of the program's execution time, unless the block is within a fre-

---

1. In general, loops can be identified by a compiler based on the dataflow graph for the input program. Although loop identification could be applied in the context of the poll point placement heuristics developed here, the current implementation of APRIL utilizes the simpler approach of identifying loops based on language constructs. In C, these consist of “**for**”, “**while**”, and “**do-while**”.

quently executed loop or function call. Thus, it seems unlikely that placing poll points in the middle of a basic block will offer improved checkpoint responsiveness, and could add overhead and hinder optimizations. Second, the primary mechanisms for induction in procedural programming are iteration (i.e. loops) and subroutine invocation (e.g. recursion). Since subroutine invocation already causes periodic polling due to mandatory poll point placement, it seems likely that the addition of optional poll points into loops could provide more complete periodic polling coverage over the lifetime of a program and thus lead to lower on-average checkpoint request wait times.

The narrowing of optional poll point placement to loops simplifies the design of a poll point placement policy, but the naive policy of placing a poll point at the end of every loop could lead to poor performance. Recall from Chapter 4 that careless placement of poll points can prevent the application of many back-end optimizations, including those performed on loops. To further restrict the placement of poll points, the APRIL heuristic policies classify loops based on the following factors:

- $N_{outer}$ , the outer nesting factor of a loop. This value is defined to be the number of loops that contain the loop in question. If the loop appears in the outermost scope of its containing function,  $N_{outer}$  is defined to be zero. If a loop is contained in one or more other loop bodies, the maximal  $N_{outer}$  value of which is  $n$ , the loop is defined to have 
$$N_{outer} = n + 1 .$$
- $N_{inner}$ , the inner nesting factor of the loop. A loop that contains no loops in its body is defined to have  $N_{inner}$  equal to zero. If a loop contains any other loops in its body, its  $N_{inner}$  value is defined to be one greater than the maximal  $N_{inner}$  of the loops it contains.
- $N_{body}$ , the number of statements contained in the loop's body.

Based on these values, we define the following loop classifications for convenience in describing our placement heuristics:

- A *nested loop* is defined to be a loop for which  $N_{inner} > 0$ .
- An *outermost loop* is defined to be a loop for which  $N_{outer} = 0$ .
- An *innermost loop* is defined to be a loop for which  $N_{inner} = 0$  and  $N_{outer} > 0$ .

Based on these classifications, the APRIL poll point placement heuristic supports options to select the loops into which poll points should be placed. The following selection criteria are implemented:

- *Policy=1*—optional poll points placed in each nested loop.
- *Policy=2*—optional poll points placed in complex outermost loops, i.e. outer loops for which  $N_{body} \geq C_{out}$  for some constant  $C_{out}$ .
- *Policy=4*—optional poll points placed in each outermost loop.
- *Policy=8*—optional poll points placed in complex innermost loops, i.e. innermost loops for which  $N_{body} \geq C_{in}$  for some constant  $C_{in}$ .
- *Policy=16*—optional poll points placed in each innermost loop.

Any combination of these policies can be selected by adding their policy numbers. For example, selecting *Policy=12* would cause poll points to be placed in all outermost and complex innermost loops. Note, the base policies are not mutually exclusive—for example, *Policy=4* will always select a superset of the points selected by *Policy=2*. In all, there are 18 unique combinations of policies including a special policy, *Policy=0*, that specifies that only mandatory poll points should be placed. All of the policies except *Policy=0* implement the mandatory poll point optimization described in Section 3.3.3 (we omit this optimization for *Policy=0* to avoid potentially factoring all poll points out of a program). We will examine the performance trade-offs associated with different policy combinations in Chapter 7.

## 6.2.2 Function Prologues

Whereas poll points give a program the ability to capture the stack and execution state using

the normal function return mechanism, function prologues as described in Chapter 3 must be added to all functions transformed by APrIL to support stack and execution state recovery using the normal subroutine call mechanism.

At the beginning of each function that it transforms, APrIL places a check to determine if a restart is in progress. Similar to the implementation of poll points, this check examines the value of the `PIL_ChkptStatus` variable exported by the PIL. To implement the case where the function is called during a process restart, APrIL generates code to restore the values of all local variables and actual parameters using the PIL stack management interface. After the local variables are recovered, the PIL reads the integer value corresponding to the poll point at which the checkpoint of this stack frame was generated. Based on this poll point number, APrIL uses a case statement to jump to the appropriate code location (i.e. poll point-related label statement) in the function. If the target of the jump is not the first label associated with a mandatory poll point, the prologue also clears the value of the `PIL_ChkptStatus` variable by calling the PIL function `PIL_DoneRestart()`, indicating that the restoring function is the last one on the call stack, and normal execution will resume after the jump to the poll point. If the target of the jump is the first label associated with a mandatory poll point, the jump will immediately be followed by a call to the next function on the stack that needs to be restored, thus allowing the normal function call mechanism to be used to recover the stack and execution state. Figure 6.3 illustrates an APrIL function prologue transformation. The function heading given in Figure 6.3(a) is transformed to include the prologue depicted in Figure 6.3(b).

Note that in addition to performing the restart-related activities described above, the prologue is also responsible for registering with the PIL the addresses of any local variables or parameters that may be referred to by pointers in the function. This additional action is required to support pointer description and resolution (as described in Section 5.3.6). For example, the function in Figure 6.3 has an array `x` whose address is used at some point in the function, and thus a call to regis-

ter the address, size, and type of this array is generated.

---

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

---

(a) The original function heading

---

```
void example(double *A) {
    int i;
    double X[100];
    PIL_RegisterLocal(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 6.3:** A function prologue.

### 6.2.3 Function Epilogues

In the model described in Chapter 3, the code executed at each poll point during a checkpoint operation is responsible for capturing the state of all local variables and actual parameters in a call frame. In order to produce more compact transformed programs, APRIL unifies the code to capture the state of a call frame into one code segment: the function epilogue. As described in Section 6.2.1, poll points generated by APRIL include code to jump to a function epilogue in the event that a checkpoint is in progress. The job of the function epilogue is to save all of the local variables and actual parameters for the function using the PIL stack management interface. 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 function's last return statement<sup>1</sup>; 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 6.3 is depicted in Figure 6.4.

We note that this design for saving the local state associated with a function call assumes 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 during its pre-processing phase, renaming where appropriate to avoid name clashes.

---

```

    return; /* End of code for normal function execution */
_PIL_save_frame_:
    PIL_InitStackFrame(PIL_LogicalPointerSize + sizeof(int));
    PIL_SaveStackPointer(A);
    PIL_SaveStackInt(i);
    PIL_SaveStackDoubles(X,100);
    return;

```

---

**Figure 6.4:** A function epilogue.

## 6.2.4 Module Initialization

The three types of transformations discussed thus far are primarily aimed at implementing the checkpoint and restoration of function call stacks. In general, to use the PIL interface correctly during these activities, APrIL must generate code to register any types defined by the program with the type description table. For example, a stack save operation may be required to capture the state of a variable of a user defined structure type. Furthermore, to support the automatic checkpoint and recovery of global variables, APrIL must register all globals defined in the program with

---

1. In the case of `void`-returning (or other, perhaps erroneous) functions that do not end with a `return` statement, APrIL generates a `return` to ensure that APrIL state capture code is never accidentally executed during normal execution.

the global variable table. APrIL generates code to perform both type table and global variable registrations in an added initialization function, which we will call `APrIL_Init()`<sup>1</sup>. This initialization function is registered as an event handler for the process startup event exported by the PIL.

The first activity performed in `APrIL_Init()` is the registration of user defined types. Recall, the PIL type description interface assigns an integer identifier to all registered types. When registering global variables, allocating memory from the heap, or performing a checkpoint, the transformed code may need to make use of these integer type identifiers. In order to make the type identifiers associated with user defined types globally accessible, APrIL introduces a global integer variable corresponding to each type<sup>2</sup>. For each user defined structure type, APrIL declares this type identifier in the global scope, and inside `APrIL_Init()` registers a type description of the structure by iterating over its fields. Internally, APrIL maintains a mapping between data types and type identifier variables to support other transformations such as function prologue and epilogue generation.

Note that if an application consists of multiple source code files, redundant type descriptions may be generated in the separately compiled files. For example, more than one program file might include the same header file, which may in turn define a structure type. In each file, a different global type identifier variable will be introduced, and in each file's initialization file, the type will be registered in the type table. However, this redundant type registration is functionally harmless, and does not introduce significant runtime overhead (we examine the performance of applications relying on separate compilation in Chapter 7).

After the type registrations generated in `APrIL_Init()`, APrIL emits global variable registrations. For each global variable defined in the input code, APrIL generates an appropriate call to

- 
1. The actual name of the generated routine is a quasi-unique string based on the input file name and size in order to support separate compilation. Many transformed files may be later linked together, in which case each of their initialization routines will be invoked.
  2. Again, to support separate compilation, the introduced variable names are quasi-unique strings based on the type description and the name and size of the file in which the type is declared.



the PIL global registration function.

## 6.2.5 Heap Allocation Transformations

As described in Section 5.3.5, the ability of the PIL to perform pointer description correctly and to provide automatic capture and recovery of heap allocated memory blocks is predicated on the use of the PIL heap management interface. To support existing C code, and the use of the more familiar standard C library heap management interface, APrIL attempts to transform calls to standard C heap routines into PIL heap routines. The primary difficulty involved in this transformation is the requirement that APrIL determine the type of the memory block allocated by each heap allocation call—the standard C dynamic memory allocation interface is untyped, but the PIL interface is based on typed allocation.

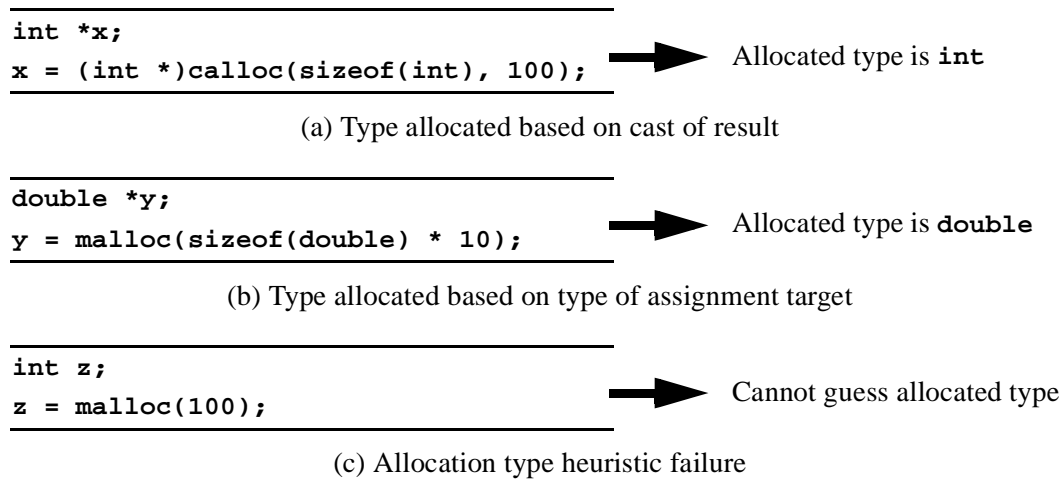
The APrIL heap allocation transformations are based on the identification of all expressions in the program deemed to be heap management expressions—basically, expressions containing calls to the C standard library routines `malloc()`, `calloc()`, `realloc()`, or `free()`. A grammar that describes heap allocation expressions as defined by APrIL is presented in Figure 6.5.

<code>&lt;heap-management-expr&gt;</code>	::	<code>&lt;alloc-assign&gt;   &lt;free-expr&gt;</code>
<code>&lt;free-expr&gt;</code>	::	<code>free ( &lt;expr&gt; )</code>
<code>&lt;alloc-assign&gt;</code>	::	<code>&lt;expr&gt; = &lt;alloc-expr&gt;</code>
<code>&lt;alloc-expr&gt;</code>	::	<code>&lt;cast-alloc-expr&gt;   &lt;uncast-alloc-expr&gt;</code>
<code>&lt;cast-alloc-expr&gt;</code>	::	<code>( &lt;type&gt; ) &lt;uncast-alloc-expr&gt;</code>
<code>&lt;uncast-alloc-expr&gt;</code>	::	<code>malloc ( &lt;expr&gt; )</code> <code>            calloc ( &lt;expr&gt; , &lt;expr&gt; )</code> <code>            realloc ( &lt;expr&gt; , &lt;expr&gt; , &lt;expr&gt; )</code>
<code>&lt;expr&gt;</code> produces a valid C expression		
<code>&lt;type&gt;</code> produces a valid C type name		

**Figure 6.5:** Heap management expression grammar.

The transformation of free expressions is straightforward—the invocation of `free()` is changed to an invocation to `PIL_Free()`, with the parameter unchanged. The transformation of allocation expressions is more difficult, since the PIL allocation routines require a type table index

indicating the type of memory allocated. To determine the memory type allocated by an allocation expression, APRIL performs the following heuristic. If the result of the allocation expression is type cast (i.e. if the allocation expression matches the *cast-alloc-expr* production in Figure 6.5), and the cast type is a pointer type, the allocated memory is assumed to be of the type referred to by the cast type, as in Figure 6.6(a). If the heap allocation fails to match this pattern, APRIL next attempts to infer the type based on the left-hand side of the heap allocation assignment<sup>1</sup>. If the type of the left-hand side of the heap allocation assignment is a pointer type, the allocated memory is assumed to be of the type referred to by this type, as in Figure 6.6(b). If the heap allocation expression fails to match the grammar defined in Figure 6.5, or if the type allocated cannot be determined using the described heuristic, as in Figure 6.6(c), APRIL compilation fails with an appropriate error message to the user.



**Figure 6.6:** Type allocation heuristic examples.

It is important to note that this heuristic for determining the type of memory allocated by a heap allocation expression can fail for certain memory allocation schemes without notification at

---

1. Recall, we have assumed for these transformations that function calls, including heap allocation calls, appear only in C expression statements. For example, a heap allocation call could not appear in a **return** statement, which is classified as a jump statement. Given this assumption, it is fair to further assume that the result of any heap allocation instruction will appear on the right hand side of an assignment so that the allocated memory can later be utilized by the program.

compile time. For example, consider a program written to make use of a wrapper function that is called throughout the program in place of `malloc()`. This wrapper function might keep track of the amount of memory allocated by the program to debug a memory leak, and then call on `malloc()` to perform actual memory allocation. The result of this call to `malloc()` in the wrapper function might be assigned to a temporary variable and returned as a pointer to type `char`, which in turn would be cast to the appropriate allocated type at the wrapper function call site. In this scenario, the type of memory actually allocated by the call to `malloc()` in the wrapper function cannot be guessed statically, as it will vary at run time. Despite this, assuming the allocation expression matches one of the acceptable patterns, APrIL will incorrectly identify the type of memory allocated to be of type `char`. Although the transformed program will still operate correctly until it is checkpointed and restarted, a restart of the process on a different platform will result in incorrect program state being restored. Given the current implementation, programmers have the options of (a) being aware of this weakness in the heuristic and coding their memory allocation scheme appropriately, or (b) calling on the PIL heap management interface directly instead of relying on the APrIL heap transformation heuristics, which in all cases will provide the safest interaction with the system.

## 6.2.6 Marshalling Functions

As described in Section 5.3.3, the PIL supports the registration of user defined marshalling functions for data types, a mechanism that can offer improved typed I/O performance. As a compile-time option, APrIL can generate and register marshalling functions for all user defined structure types.

To generate marshalling functions, APrIL iterates over the structure types defined in the input program. For each of these types, APrIL generates two new functions: a *pack* routine to output memory regions containing the type to a PIL buffer, and an *unpack* routine to input values of the

type from a PIL buffer. The definitions of these routines contain calls to the appropriate typed I/O functions for each field in the structure. In the case of fields of basic data types, the PIL buffer interface is called on directly.

## 6.3 Pre-processing

As described at the beginning of Section 6.2, the fundamental APrIL transformations operate under a set of simplifying assumptions. The restrictions that these assumptions place on the allowable input code would seriously detract from the usability of APrIL for real applications—most existing C programs would almost certainly be disallowed. To avoid these restrictions, APrIL contains a pre-processor module that performs a set of semantics-preserving transformations on the input code to enforce the assumptions made by the later transformations, allowing a more general set of input programs. In this section, we examine the fundamental transformations performed by the pre-processor. It should be noted that these transformations are not directly related to state capture or recovery. However, in practice, the pre-processor transformations are critical for automatically factoring non-trivial programs into a subset of ANSI C that is simple enough to serve as the intermediate representation for Process Introspection transformations.

### 6.3.1 Variable Declaration Motion

To support the APrIL function epilogue strategy for capturing the state of a function invocation, all local variable declarations are assumed to appear in the outermost scope of their function definition. In general, local variable declarations can appear at the beginning of any compound statement in C (i.e. any list of statements enclosed by the “{” and “}” tokens). To allow this feature of C, and still meet the assumptions of the APrIL transformations, the APrIL pre-processor moves the declarations of all local variables that do not appear in their containing function’s outermost scope. To avoid name clashes with other variables visible in the function’s scope, APrIL renames

moved variables. After the variable is renamed, all references to the replaced variable name in its previous inner scope are located and transformed to refer to the new identifier.

### 6.3.2 Function Call Motion

One of the assumptions made by the APrIL mandatory poll point placement transformation is that function calls appear only in C expression statements—statements containing a valid C expression followed by a semicolon. This precludes function calls from appearing in locations such as iteration statements (e.g. **while**, **for** loops), selection statements (e.g. **if**, **case** statements), and so on. In general, functions can appear in any valid C expression, which in turn can appear in a number of contexts that are not expression statements.

To support general C input code, the APrIL pre-processor moves function calls that are found in unacceptable locations. The implementation of this transformation scans the statements of the input program, locating statements that contain one or more expressions containing any function calls, and which are not C expression statements. When such a statement is found, it is transformed by moving the evaluation of any function-calling expressions out of the statement and into introduced expression statements, in which they are assigned to introduced temporary variables. In the transformed statement, the temporary variables are used in place of the corresponding moved expressions.

The details of this transformation depend on whether the statement in question is an iteration statement (i.e. a loop construct). The simpler case is that of non-iteration statements, as depicted in Figure 6.7, in which the transformation of an **if** statement is depicted. In this transformation, the expression that contains a function call is assigned to a temporary variable in an introduced expression statement immediately preceding the transformed statement. In the original **if** statement, the new temporary variable is used in place of the moved expression. The same transformation is used for other non-iteration statement types, such as **return** and **switch**.

<pre> if (&lt;expr&gt;) {     . . . } </pre>	<pre> tmp = &lt;expr&gt;; if (tmp) {     . . . } </pre>
(a) Original code. Assume that <b>expr</b> contains one or more function calls.	(b) Transformed code

**Figure 6.7:** Function call motion, **if** statement example.

The transformation of iteration statements (i.e. **for**, **while**, and **do-while**) is somewhat more complicated. Each type of iteration statement contains a conditional expression that must be evaluated to test for termination before each iteration is performed. This expression must be evaluated and its result must be assigned to a temporary variable in three types of locations: (1) before the iteration statement is reached, (2) at the end of the loop body (i.e. before the next iteration is performed), and (3) before any **continue** statement for the loop is executed. In addition to the conditional test, **for** loops contain an initialization expression that must be assigned to a temporary variable before the **for** statement is reached, and an increment expression that must be executed before the conditional test for each loop iteration. An example of the function call motion transformation on a **for** loop is presented in Figure 6.8. Similar transformations are implemented for **while** and **do-while** statements.

<pre> for(&lt;expr1&gt;; &lt;expr2&gt;; &lt;expr3&gt;) {     . . .     continue;     . . . } </pre>	<pre> &lt;expr1&gt;; tmp1 = &lt;expr2&gt;; for(; tmp1; ) {     . . .     &lt;expr3&gt;; tmp1 = &lt;expr2&gt;;     continue;     . . .     &lt;expr3&gt;; tmp1 = &lt;expr2&gt;; } </pre>
(a) Original code. Assume that <b>expr1</b> , <b>expr2</b> , and <b>expr3</b> each contain function calls.	(b) Transformed code

**Figure 6.8:** Function call motion, **for** loop example.

### 6.3.3 Function Call Separation

After the transformations described in Section 6.3.2 have been performed, all function calls will appear exclusively in C expression statements, but in general, each expression statement may contain any number of function calls. However, recall that one of the assumptions made due to the implementation of APrIL mandatory poll points is that at most one function call can appear in any single C expression statement. To meet this requirement, the APrIL pre-processor separates C expression statements containing multiple function calls into a sequence of C expression statements, each containing at most one function call.

The separation of function calls within a C expression statement is performed using a recursive algorithm. When the transformation is applied to a given expression statement, it factors that expression into new expression statements to evaluate constituent subexpressions. Each of the subexpressions may in turn contain multiple function calls, in which case the transformation is applied recursively to its expression statement. The actual separation of individual expressions depends on their expression variety. For example, for binary operations, the left hand side is assigned to a temporary variable in an introduced preceding expression statement, and the transformed binary operation consists of the temporary variable on the left hand side, and retains its original right hand side.

An example of the recursive application of the APrIL function call separation transformation algorithm is presented in Figure 6.9. The highest level expression variety in the original expression statement depicted in Figure 6.9(a) is an assignment operation. The function separation algorithm is applied to this statement, and introduces a new expression statement assigning the right hand side of the assignment to a temporary variable, depicted in Figure 6.9(b). This introduced expression statement is an addition operation, which must itself be subject to function call separation, the result of which is depicted in Figure 6.9(c). Note, each of the three resulting statements may still contain multiple function calls, in which case the recursion would continue. When the algorithm is

applied to an expression containing zero or one function calls, the recursion halts.

---

```
<expr1> = <expr2> + <expr3>;
```

---

(a) Original code. Assume `expr1`, `expr2`, and `expr3` each contain function calls.

---

```
tmp1 = <expr2> + <expr3>;
<expr1> = tmp1;
```

---

(b) Transformation applied to the outermost expression, an assignment.

---

```
tmp2 = <expr2>;
tmp1 = tmp2 + <expr3>;
<expr1> = tmp1;
```

---

(c) Transformation applied to a contained expression, an addition operation.

**Figure 6.9:** Function call separation example.



# Chapter 7

## Applications and Performance Results

The Process Introspection system described in Chapter 5 and Chapter 6 has been implemented and has been tested on a set of significant applications. In this chapter, we discuss the results of detailed performance study conducted on the system using these applications.

### 7.1 Performance Metrics

A fundamental question regarding the performance of any system is which metrics will provide an accurate picture of the costs and benefits associated with services provided by the system. In this section we describe the reasoning behind the set of performance metrics used to evaluate the overhead and efficiency of the Process Introspection system.

#### 7.1.1 Performance Overhead

For the Process Introspection system, the impact on basic performance caused by the system (i.e. the amount that it slows down an application when process state capture and recovery activities are not taking place) is of primary interest. Ideally, users should not have to pay anything for a service when that service is not in use, but the nature of the Process Introspection mechanism implies some level of overhead for applications even when state capture and recovery services are not in use. Activities such as type and global variable registration and the execution of poll points can add significant run-time overhead to the process, regardless of the frequency with which the state capture facility is used. Two metrics of interest with respect to performance overhead are:

- *Introduced overhead*—The Process Introspection transformations add extra instructions to a program. For example, periodically the program must poll for checkpoint requests. During heap allocation, a record of the allocated block must be added to the heap allocation table. We can examine the introduced overhead by comparing the execution time of an application with

and without the Process Introspection transformations applied. In the case of the transformed program execution, no checkpoints are requested and no restarts are performed. The slowdown of the transformed code provides a measure of the overhead introduced by Process Introspection. For a given program, we define  $t_{norm}$  to be the execution time of the non-transformed version of the program compiled without optimizations. The execution time of that program transformed using a poll point placement policy  $p$ , and compiled without optimizations will be called  $t_{trans,p}$ . The overhead introduced by the Process Introspection transformations, expressed in terms of percentage of the non-transformed program's execution time is defined to be:

$$O_{norm,p} = (t_{trans,p} - t_{norm}) / t_{norm} \quad (7.1)$$

- *Impact on optimizations*—As was described in Chapter 4, the Process Introspection transformations can have significant impact on compiler optimizations. In order to measure the impact of the transformations on the effectiveness of compiler optimizations, we can examine the performance of an application compiled with and without the Process Introspection transformation, in each case with back end optimizations applied. As above, we can define the overhead introduced by Process Introspection on an optimized version of the program. Let the execution time of the non-transformed, optimized code be  $t_{opt}$ . The execution time of the program transformed using poll point placement policy  $p$ , and compiled with optimization is  $t_{topt,p}$ . The overhead introduced by the Process Introspection transformations into the optimized program, expressed in terms of percentage of the non-transformed optimized program's execution time is defined to be:

$$O_{opt,p} = (t_{topt,p} - t_{opt}) / t_{opt} \quad (7.2)$$

In some senses, this metric is the most important measurement of the impact of Process Introspection, as it defines how much slower an end-user application will actually run. On the other

hand, this measure fails to capture the impact of Process Introspection on optimizations. Given that an application runs a given amount slower, how much of this slowdown is due to extra work introduced by Process Introspection, and how much is due to hindrance of the optimizer? We can quantify the impact on the optimizer in terms of the fraction of obtainable speedup achieved. The speedup obtained by the optimizer on the program without Process Introspection transformations is defined to be:

$$S_{ntrans} = t_{norm}/t_{opt} \quad (7.3)$$

Similarly, the speedup the optimizer achieves for the transformed version of the program is:

$$S_{trans,p} = t_{trans,p}/t_{opt,p} \quad (7.4)$$

Thus, the fraction of the “available” speedup achieved by the optimizer on the transformed code, which we call the *effectiveness* of the optimizer is:

$$E_p = S_{trans,p}/S_{norm} \quad (7.5)$$

Ideally, we would like  $O_{norm,p} \approx 0$  and  $E_p \approx 1$ , and thus  $O_{opt} \approx 0$ . In practice, we expect that some overhead is unavoidable. However, we have stated as a design goal that  $O_{opt}$  be no more than 10%.

As a hypothetical example, consider a program for which  $t_{norm} = 10$  and  $t_{opt} = 5$ . We transform the program using APrIL with poll point placement policy 3, and find that  $t_{trans,3} = 11$ , and  $t_{opt,3} = 7$ . In this case, the introduced overhead associated with Process Introspection  $O_{norm,3} = 0.1$ , or 10%. The impact on the effectiveness of program optimizations,  $E_p$ , is 0.79—i.e. the optimizer is only about 79% as effective as it could have been in the absence of Process Introspection. The total overhead on the optimized program is  $O_{opt,3} = 0.4$ , or 40%.

### 7.1.2 State Capture and Recovery Costs

A second equally important class of performance metrics measures the performance of the

state capture and recovery mechanism itself. Two key metrics in this class are:

- *Checkpoint request wait time*—If a checkpoint is requested from a process, we would ideally like that process to begin constructing a checkpoint immediately. In the Process Introspection mechanism, we must wait until a poll point is reached in order to start constructing a checkpoint, and so checkpoint requests will suffer a certain wait time before being serviced. We can obtain a measure of this wait time by measuring the average time between poll points for a given program transformed using a given poll point placement policy. Assuming a uniform distribution of probability of checkpoint requests arriving over time, checkpoint requests will be forced on average to wait half the average interval between poll points. We can obtain an estimate of the average interval between poll points by counting the number of poll points encountered during a program’s execution. Given a poll point count  $n_p$  for an execution of a program transformed with poll point placement policy  $p$ , and assuming that the optimized version of the program will be executed, the average poll point interval for the program is given by:

$$I_p = t_{opt,p}/n_p \quad (7.6)$$

- *State capture and recovery costs, both time and space*—Ideally, we would like the time required to construct a checkpoint or to perform a restart to be as small as possible, and we would like the resulting checkpoint to be as small as possible. In practice, the time to perform a state capture or recovery using Process Introspection is largely dictated by the size and complexity of the process state in question and by the performance characteristics of the medium to which the process state is written or from which it is read. Besides raw performance measurements (i.e. time to perform state capture and recovery, size of captured state), a performance metric of interest with respect to Process Introspection state capture is the overhead introduced into the state recovery mechanism due to heterogeneous restarts. Recall, Process Introspection as implemented employs a receiver-makes-right data conversion strategy. Thus,

whereas the time to perform a state capture will not vary in heterogeneous environments, the time to perform state recovery will generally be greater if a restart is performed on a node with a data type incompatible with that of the checkpoint process. The percentage of overhead introduced by heterogeneous restarts is thus an important performance characteristic of our implementation.

## 7.2 Experimental Setup

Our performance experiments were run on a heterogeneous set of computer systems, and using a heterogeneous set of back end compilers. Besides being necessary to perform heterogeneous restart experiments, a mixture of processor types is important to evaluate the sensitivity of the Process Introspection transformations to different underlying architectural features. For example, the impact of the poll points inserted by the APrIL compiler might be greater on a processor that predicts “branch not taken” at poll point conditional branches—the common case at a poll point is to take the branch, bypassing the code required to service a checkpoint request. A heterogeneous set of optimizing back-end compilers is important to evaluate the impact of the Process Introspection transformations on optimizations for different optimizer implementations. The set of test platforms used for the experiments described in this section is listed in Table 7.1. In the remainder of this section, we refer to these systems by their name as listed in the left-most column of Table 7.1.

<i>Name</i>	<i>Processor type</i>	<i>Processor speed</i>	<i>Memory</i>	<i>Operating system</i>	<i>Compiler</i>
<b>alpha</b>	DEC alpha	500 Mhz	128 MB	Linux 2.0	GNU gcc 2.7.2
<b>x86</b>	Intel Pentium	200 Mhz	64 MB	Linux 2.0	GNU gcc 2.7.2
<b>rs/6000</b>	PowerPC 601	80 Mhz	128 MB	AIX 4.2	xlc 3.1
<b>mips</b>	MIPS R4000	100 Mhz	64MB	IRIX 6.2	SGI cc
<b>sparc</b>	Sparc20	50 Mhz	512 MB	SunOS 5.5.1	Sun cc 3.0

**Table 7.1:** Test platforms.

### 7.3 Basic Numerical Applications

Initially, to examine the performance characteristics of our prototype implementation, we applied the system to a set of basic numerical applications. The programs in this basic test suite average approximately 120 lines of code each, and do not rely on separate compilation (i.e. each program is contained in a single source file). The individual programs in the set are:

- *mm* (Matrix Multiply)—Computes the product of two dense, square matrices of 256x256 double precision floating-point numbers using the standard  $O(n^3)$  algorithm.
- *gs* (Gauss-Seidel)—Solves the sparse linear system of  $10^4$  equations resulting from the discretization of a two dimensional Poisson equation with Dirichlet boundary conditions. The algorithm used is a standard Gauss-Seidel five point stencil iteration applied to a 80x80 grid of solution elements until the change in the two-norm of the solution is less than  $10^{-2}$ .
- *qs* (Quicksort)—Applies a standard quicksort algorithm to an array of  $2^{21}$  integers.
- *ge* (Gaussian Elimination)—Performs Gaussian elimination with partial pivoting on a dense 512x512 matrix, followed by a back-substitution phase to obtain the solution vector.
- *cg* (Conjugate-Gradient)—Applies a basic conjugate-gradient iteration (no preconditioning) to the same linear system solved by the Gauss-Seidel test, using the same convergence criterion as that example, with the solution discretized onto a 200x200 grid.

Each of these programs was transformed using a range of poll point policies, and was compiled for each of the test platforms with and without optimizations. We measured the mean execution time over 16 trials for each version of the programs on each of the test platforms, and based on these observations, we computed values of  $O_{norm,p}$ ,  $O_{opt,p}$ , and  $E_p$  for each policy on each platform (based on Equation 7.1, Equation 7.2, and Equation 7.5, respectively). The full details of the results of these experiments are presented in Appendix A. We now examine salient general results from the study.

In Table 7.2 we present the average  $O_{norm,p}$  observed over all test platforms for each program.

Recall from Section 6.2.1.1, higher numbered poll point placement policies generally involve more aggressive placement of optional poll points, and thus we observe that higher numbered policies generally introduce a greater percentage of overhead. For example, policy 0 which places only mandatory poll points always introduces very low overhead, whereas policy 21 which places a poll point as the last statement of each loop body generally introduces an unacceptably high degree of overhead. Intermediate policies generally perform well, introducing less than 10% overhead, but note that their effect is application dependent. Since these heuristic policies base their placement on simple loop body classifications (see Section 6.2.1.1), the overhead associated with these policies is highly dependent on loop granularity. For example, the quicksort program executes more iterations of finer grain loops than does the gaussian elimination example. Thus, all but policy 0 introduce a significant amount of overhead into the quicksort example.

	<i>mm</i>	<i>gs</i>	<i>qs</i>	<i>ge</i>	<i>cg</i>
$O_{norm,0}$	2.3%	0.7%	3.2%	4.0%	4.2%
$O_{norm,1}$	2.4%	2.1%	9.4%	4.3%	3.0%
$O_{norm,2}$	3.0%	1.7%	9.0%	3.6%	4.6%
$O_{norm,3}$	4.1%	2.4%	9.6%	4.1%	3.0%
$O_{norm,4}$	2.2%	10.6%	9.6%	3.2%	19.6%
$O_{norm,5}$	4.1%	10.3%	10.3%	3.3%	19.9%
$O_{norm,11}$	6.4%	1.8%	11.0%	4.4%	3.4%
$O_{norm,13}$	7.0%	10.4%	11.8%	3.9%	19.5%
$O_{norm,21}$	12.6%	15.9%	29.1%	14.1%	22.2%

**Table 7.2:** Basic applications, average  $O_{norm}$ .

Thus far, we have only considered overhead introduced into non-optimized versions of the programs. In practice, the performance of optimized code is of greater interest in high-performance computing settings. In Table 7.3 we present the average optimizer effectiveness ( $E_p$  as defined by Equation 7.5) over all test platforms for each application. Recall, this metric indicates the speedup achieved by the optimizer for a transformed version of a program compared to the speedup achieved for a non-transformed version of the program. As expected, the effectiveness of

the optimizer is determined largely by the aggressiveness of the poll point placement policy. For example, a conservative placement policy such as 1, which places optional poll points only in outer nested loops, has little measurable effect on the optimizer performance, whereas an aggressive placement policy such as 21 reduces the achieved speedup by up to 20%. As in the case of introduced overhead, the compiler effectiveness is application dependent. For example, the quick-sort program suffers serious reductions in speedup for all but the most conservative placement policy, whereas the conjugate gradient example achieves nearly no reduced speedup for any of the placement policies. It is worth noting that in some cases,  $E_p > 1$  is observed. This observation simply indicates that slightly better speedup was achieved for the transformed code than was achieved for the non-transformed code. Since the transformations introduce additional code that is also subject to optimizations, and more generally since the transformations change the instruction sequence manipulated by the optimizer, this is not an implausible result. These points indicate cases where the optimizer was able to achieve good speedup and reduce the overhead of the transformations, and should not be confused with the relatively unlikely possibility of observing a speedup due to the Process Introspection transformations.

	<i>mm</i>	<i>gs</i>	<i>qs</i>	<i>ge</i>	<i>cg</i>
E <sub>0</sub>	1.01	0.99	0.95	1.03	1.00
E <sub>1</sub>	1.00	1.00	0.89	1.03	0.98
E <sub>2</sub>	1.00	0.99	0.88	1.02	0.99
E <sub>3</sub>	0.98	0.99	0.89	1.01	0.99
E <sub>4</sub>	0.97	0.96	0.89	1.01	0.97
E <sub>5</sub>	0.98	0.95	0.88	0.99	0.97
E <sub>11</sub>	1.00	0.99	0.89	1.03	0.99
E <sub>13</sub>	1.03	0.95	0.90	1.02	0.96
E <sub>21</sub>	0.83	0.80	0.84	0.86	0.97

**Table 7.3:** Basic applications, optimizer effectiveness.

The metrics discussed thus far indicate the amount of overhead introduced by the Process Introspection transformations and the effectiveness of optimizations in the presence of these transformations. However, the most important raw performance metric is slowdown of the optimized



version of a program—put simply, how much slower will a program execute due to both the introduced overhead and the impedance of optimizations. In Table 7.4, we present the average over all test platforms of the overhead introduced into optimized versions of the programs, as defined by Equation 7.2. In this table, points of good performance indicate cases where both the introduced overhead was low, and the speedup achieved by the optimizer was not seriously reduced. As would be expected, since both overhead and optimizer effectiveness were functions of the placement policies aggressiveness, so is the overhead introduced into optimized programs. For example, policy 0 combines low overhead with low impact on optimizations, and thus generally performs quite well. Policy 21 combines high overhead with serious impact on optimizations, which results in unacceptably high net overheads around 50%. As in previous cases, the net overhead is also application

	<i>mm</i>	<i>gs</i>	<i>qs</i>	<i>ge</i>	<i>cg</i>
$O_{opt,0}$	1.7%	1.5%	8.4%	1.1%	4.1%
$O_{opt,1}$	2.2%	2.5%	24.1%	1.0%	5.3%
$O_{opt,2}$	2.8%	3.3%	25.7%	1.1%	5.8%
$O_{opt,3}$	6.5%	3.2%	24.2%	3.5%	4.3%
$O_{opt,4}$	5.5%	15.0%	25.0%	2.2%	23.5%
$O_{opt,5}$	7.0%	16.2%	27.2%	5.3%	22.9%
$O_{opt,11}$	6.5%	2.9%	25.6%	1.2%	4.3%
$O_{opt,13}$	3.7%	16.1%	25.7%	2.2%	23.7%
$O_{opt,21}$	54.8%	47.8%	57.1%	40.3%	26.0%

**Table 7.4:** Basic applications, average  $O_{opt}$ .

dependent. For example, policy 4 achieves relatively low overhead for the matrix multiply example, but has somewhat less desirable impact on the gauss-seidel example.

Table 7.4 presents overheads averaged over all platforms, providing insight into the average impact that each poll point placement policy will introduce into each program. The degree to which this overhead depends on the platform is also of interest, as different architecture and compiler pairs may result in different levels of overhead for the same transformed code. In Table 7.5 we present the net overhead introduced into optimized versions of the programs (defined by Equ-

tion 7.2) averaged over all applications for each test platform. We do in fact find that the introduced overhead is dependent on the platform. For example, while the rs/6000 generally suffers high overhead for all but the most sparse poll point placement policy, the x86 platform generally results in low introduced overhead.

	<i>alpha</i>	<i>x86</i>	<i>rs/6000</i>	<i>mips</i>	<i>sparc</i>
$O_{opt,0}$	1.7%	1.7%	8.7%	3.9%	0.8%
$O_{opt,1}$	6.2%	3.0%	14.8%	8.7%	2.4%
$O_{opt,2}$	6.2%	3.1%	14.9%	11.4%	3.1%
$O_{opt,3}$	6.6%	3.1%	14.9%	11.5%	5.6%
$O_{opt,4}$	9.5%	8.3%	17.0%	23.8%	12.6%
$O_{opt,5}$	11.2%	8.1%	17.1%	27.1%	15.1%
$O_{opt,11}$	8.2%	2.6%	15.0%	11.4%	3.1%
$O_{opt,13}$	10.9%	8.0%	17.5%	23.1%	12.0%
$O_{opt,21}$	20.6%	12.5%	44.7%	91.4%	56.8%

**Table 7.5:** Basic applications, per-platform average  $O_{opt}$ .

Table 7.4 indicates that for all test applications, one or more poll point placement strategies introduce low overhead into the program's normal execution. However, low overhead is generally only obtainable with less aggressive poll point placement strategies. This leads to the fundamental question of whether those strategies that introduce low overhead can also provide acceptably small poll point intervals. In Table 7.6 we show the poll point counts for each of the applications, run with each poll point placement strategy. These counts were obtained using versions of the programs instrumented to increment a counter at each poll point<sup>1</sup>.

<b>Policy</b>	<i>mm</i>	<i>gs</i>	<i>qs</i>	<i>ge</i>	<i>cg</i>
<i>0</i>	1	11685	18874363	7	2223
<i>1</i>	65793	461596	32129457	132868	54530
<i>2</i>	65535	473838	32129457	2052	54802
<i>3</i>	131329	473838	32129457	132868	54802
<i>4</i>	65793	38786389	34226609	3076	56204979
<i>5</i>	131329	38786389	34226609	133892	56204979
<i>11</i>	131329	473838	32129457	132868	54802
<i>13</i>	131329	38786389	34226609	133892	56204979
<i>21</i>	16908545	74335201	95791100	45528068	66946875

**Table 7.6:** Basic applications, poll point counts.

Based on the poll point counts for each program, we can calculate the average poll point interval for each using Equation 7.6. These results are presented for the x86 architecture in Table 7.7.

<b>Policy</b>	<b><i>mm</i></b>	<b><i>gs</i></b>	<b><i>qs</i></b>	<b><i>ge</i></b>	<b><i>cg</i></b>
<i>0</i>	3.7 sec.	0.42966	0.00026	1.2 sec.	5.16433
<i>1</i>	0.05548	0.01090	0.00016	0.06219	0.21060
<i>2</i>	0.05606	0.01063	0.00016	4.02885	0.20898
<i>3</i>	0.02803	0.01057	0.00017	0.06234	0.20863
<i>4</i>	0.05580	0.00015	0.00016	2.68878	0.00023
<i>5</i>	0.02769	0.00015	0.00016	0.06175	0.00022
<i>11</i>	0.02763	0.01058	0.00016	0.06215	0.20783
<i>13</i>	0.02786	0.00015	0.00015	0.06173	0.00023
<i>21</i>	0.00022	0.00008	0.00006	0.00018	0.00019

**Table 7.7:** Basic applications, average poll point intervals on x86.  
(times in milliseconds)

In most cases the intervals in Table 7.7 are quite low—10’s of microseconds or less (recall, performing state capture will typically involve communicating the process state over a network or writing it to stable storage—either of these will require orders of magnitude more time than this). However, in some cases—for example, policy 0 on the matrix multiply and Gaussian elimination examples—the average interval is quite high (or, equivalently, the number of poll points encountered is very low). In these examples, the characteristics of the application were a poor match with the placement heuristic selected. In both cases, the “mandatory poll points only” heuristic was applied, but the programs performed subroutine invocations very infrequently. Although these versions of the programs suffered almost no introduced overhead, the resulting poll-point intervals are unacceptably high. Despite the existence of points in Table 7.7 at which poll point placement resulted in poor performance, it is important to recognize that for each application, there is at least one poll point placement policy for which both very low overhead is introduced and for which the poll point interval is very low. This result is important—in all cases, we found that there was at

- 
1. An interesting note not related to performance: the correctness arguments discussed in Chapter 4 predict that all low-level versions of the program (i.e. for each platform, compiled with and without optimization) must encounter the same number of poll points during execution. We verified this result exhaustively for all applications, over all platforms, with and without optimization.

least one poll-point policy that could be selected to achieve both low overhead and low poll point intervals, and thus Process Introspection could be applied effectively. Furthermore, we found that in all cases, relatively conservative placement was acceptable—either mandatory-only placement or placement in only a few coarse grained, outer loops appeared to be sufficient.

Our final performance measurements for this application set characterize the costs of actual state capture and recovery for the test applications. In Table 7.8 we present for each application the captured state sizes, the time to perform state capture on the x86 platform, and the time to perform state recovery on the x86 platform using checkpoints generated on each of the available platforms.

	<i>mm</i>	<i>gs</i>	<i>qs</i>	<i>ge</i>	<i>cg</i>
<i>Chckpt size</i>	1573018	102613	8388716	2118390	1280353
$t_{chckpt}$	47.31	4.05	303.72	115.54	46.57
$t_{restart}$	99.84	6.13	570.50	167.34	84.60
$t_{restart,alpha}$	97.82	6.11	560.69	167.69	84.60
$t_{restart,rs/6000}$	131.70	7.51	748.07	191.12	117.61
$t_{restart,mips}$	130.62	7.41	751.45	191.79	113.15
$t_{restart,sparc}$	131.65	7.57	748.84	188.44	115.48

**Table 7.8:** Basic applications, time to checkpoint/restart on x86.  
(sizes in bytes, times in milliseconds)

First, it should be noted that as expected, state capture and recovery performance are a function of the process state size. For example, the time to capture and recover the state of the Gaussian elimination program is roughly twice the time to capture and recover the state of the conjugate gradient program, and there is approximately a factor of two difference in their state sizes.

A primary question addressed by these measurements is the price of heterogeneous state recovery. As expected, heterogeneous state recovery (for example, in this case recovering from the mips platform which uses big endian byte ordering, as opposed to recovering from the alpha platform which uses the x86-compatible little endian ordering) does add overhead. In fact, the overhead is significant, roughly 40% in some cases. Note, besides this result being important for making placement or migration decisions, this result substantiates our intuitive claim that receiver-

makes-right data coercion can improve performance. If we had used XDR, for example, all of the produced checkpoints would have used big-endian ordering, and thus all would have performed approximately as well as the incompatible recovery operations listed in Table 7.8.

## 7.4 NAS Benchmark Kernels

As a second, more substantial set of applications, we applied the Process Introspection to a subset of the NAS benchmark kernels, described in [5]. The four kernels that we used average approximately 370 lines of code each, and like the programs described in Section 7.3, do not rely on separate compilation. The kernels selected were chosen due to the availability of versions programmed in ANSI C, as required by the APrIL compiler:

- *nas-is*—Performs 10 iterations in which  $2^n$  random integer keys in the range  $[0, 2^m)$  are ranked using a linear bucket sort. Results are reported for  $n = 21$  and  $m = 11$ .
- *nas-ep*—Executes  $2^n$  iterations of a loop in which a pair of values,  $(x, y)$ , is selected from the range  $[-1, 1]$ . For each pair, if the inequality  $t = x^2 + y^2 \leq 1$  is satisfied, independent Gaussian deviates  $X$  and  $Y$  with mean zero and variance one are generated by the following:

$$X = x \sqrt{\frac{(-2 \log t)}{t}} \quad \text{and} \quad Y = y \sqrt{\frac{(-2 \log t)}{t}}$$

After all such pairs are generated, the number of pairs lying in unit width square annuli centered at the origin (within 10 units per dimension) are tabulated and output along with  $\Sigma X$  and  $\Sigma Y$  over all  $X$  and  $Y$ . Results are reported for  $n = 21$ .

- *nas-mg*—Executes four iterations of the V-cycle multigrid algorithm to obtain an approximate solution,  $u$ , to the discrete Poisson problem  $\nabla^2 \cdot u = v$  on an  $n \times n \times n$  grid with periodic boundary conditions. Results are reported for  $n = 64$ .
- *nas-cg*—Estimates the largest eigenvalue of a symmetric positive definite sparse matrix,  $A$ , with a random pattern of non-zero elements. The basic algorithm employed is:

```

 $x = [1, 1, \dots, 1]^T$ 
do  $i = 1, n$ 
    Solve the system  $Az = x$  and return  $\|r\|$  using a conjugate gradient method
     $\zeta = \lambda + 1/(x^T z)$ 
    Output  $i, \|r\|$ , and  $\zeta$ 
     $x = z/\|z\|$ 
od

```

This process results in the computation of the eigenvalue estimate,  $\zeta$ , and the residual,  $\|r\|$ , on each iteration. Results are reported for a matrix size of 1400x1400, with 120000 non-zero entries, over 15 iterations (i.e.  $n = 15$ ).

We performed the same set of experiments with the NAS benchmark kernels as were performed with the basic applications described in Section 7.3. The first set of experiments characterize introduced overhead, optimizer effectiveness, and net resulting overhead introduced into the optimized versions of the programs. The results of these experiments, averaged over all platforms, are presented in Table 7.9, Table 7.10, and Table 7.11, respectively.

	<i>nas-is</i>	<i>nas-ep</i>	<i>nas-mg</i>	<i>nas-cg</i>
$O_{\text{norm},0}$	7.5%	6.2%	3.1%	0.6%
$O_{\text{norm},1}$	7.4%	6.4%	2.7%	2.4%
$O_{\text{norm},2}$	8.8%	4.2%	1.2%	0.3%
$O_{\text{norm},3}$	8.9%	4.9%	2.7%	1.3%
$O_{\text{norm},4}$	15.1%	8.8%	1.9%	2.9%
$O_{\text{norm},5}$	15.2%	7.0%	3.6%	3.0%
$O_{\text{norm},11}$	8.8%	5.3%	1.8%	10.0%
$O_{\text{norm},13}$	14.9%	8.3%	2.4%	11.6%
$O_{\text{norm},21}$	14.7%	7.9%	3.2%	12.9%

**Table 7.9:** NAS benchmarks, average  $O_{\text{norm}}$ .

	<i>nas-is</i>	<i>nas-ep</i>	<i>nas-mg</i>	<i>nas-cg</i>
E <sub>0</sub>	1.06	0.95	0.96	0.99
E <sub>1</sub>	1.06	0.99	0.99	1.00
E <sub>2</sub>	1.02	0.95	0.97	0.98
E <sub>3</sub>	1.02	0.95	0.98	0.97
E <sub>4</sub>	0.98	1.00	0.98	1.00
E <sub>5</sub>	0.98	1.00	0.98	0.98
E <sub>11</sub>	0.99	0.96	0.93	0.87
E <sub>13</sub>	0.97	1.01	0.97	0.87
E <sub>21</sub>	0.98	1.01	0.92	0.86

**Table 7.10:** NAS benchmarks, optimizer effectiveness.

	<i>nas-is</i>	<i>nas-ep</i>	<i>nas-mg</i>	<i>nas-cg</i>
O <sub>opt,0</sub>	1.5%	12.7%	6.9%	2.0%
O <sub>opt,1</sub>	1.6%	8.4%	4.0%	2.0%
O <sub>opt,2</sub>	7.1%	9.6%	4.7%	2.8%
O <sub>opt,3</sub>	7.0%	10.4%	5.1%	4.1%
O <sub>opt,4</sub>	18.7%	9.1%	4.6%	2.8%
O <sub>opt,5</sub>	18.4%	8.0%	6.1%	5.4%
O <sub>opt,11</sub>	10.3%	10.5%	9.6%	28.5%
O <sub>opt,13</sub>	18.9%	7.9%	5.3%	30.5%
O <sub>opt,21</sub>	18.4%	7.6%	12.3%	34.2%

**Table 7.11:** NAS benchmarks, average  $O_{opt}$ .

The results of these experiments are similar to those presented in Section 7.3. Once again we find that more aggressive poll point placement generally leads to higher introduced overhead, decreased optimizer effectiveness, and thus greater slowdown of the transformed optimized code. Furthermore, we again find that in all cases, one or more placement policies achieve net overheads of less than 10%. However, there are some seemingly counterintuitive points worth noting in these results. Note that for the *nas-ep* and *nas-mg* programs, policy 0 (which we intuitively expect to be the least aggressive policy) introduces more overhead than other higher numbered cases. This seemingly unexpected result is explained by the characteristics of the transformed code. Recall from Section 6.2.1.1, all poll point placement policies employ the mandatory poll point optimiza-

tion described in Section 3.3.3, except policy 0. These programs each contain a number of simple subroutines into which many of the policies would not introduce poll points. Thus, for many of the policies, unnecessary mandatory poll points are eliminated, but for policy 0 these poll points are retained. Thus, we have the result that policy 0 as implemented may introduce more poll points than higher numbered policies.

This result is highlighted in Table 7.12, which lists the measured poll point counts for each of the benchmarks—note the poll point counts for policy 0 compared to policy 1 for the *nas-ep* and *nas-mg* kernels. We can also note that similar relationships hold for other policies—although higher numbered policies generally introduce less poll points, careful examination of their definitions confirms that for programs with certain structures the inverse relationship can also hold.

Based on the poll point counts presented in Table 7.12, we can compute the average poll point intervals for the NAS benchmarks using Equation 7.6. These results are presented for the x86 architecture in Table 7.13. Note, as in the case of the applications presented in Section 7.3, the intervals are generally quite small, but points of relatively poor performance do exist. For example, consider policy 2 applied to the *nas-mg* kernel. However, as in the case of Section 7.3, for each application studied, one or more poll point placement policies results in both low introduced overhead and small poll point intervals. Furthermore, we find that these results confirm the observation that relatively conservative poll point placement generally performs well. For example, in all cases, policy 1 results in less than 10% net overhead, and at the same time results in poll point intervals below 0.01 milliseconds.



Policy	<i>nas-is</i>	<i>nas-ep</i>	<i>nas-mg</i>	<i>nas-cg</i>
0	8388628	12582925	3858096	52976
1	8388628	4194311	136955	615785
2	10485859	4194312	2951	283908
3	10485859	4194312	137575	842502
4	73441369	4194332	11763	4767261
5	73441369	4194332	137575	4778455
11	10485859	4194312	1111343	31599790
13	73441369	4194332	1111343	35535743
21	73441369	4194332	7277331	36164967

**Table 7.12:** NAS benchmarks, poll point counts.

Policy	<i>nas-is</i>	<i>nas-ep</i>	<i>nas-mg</i>	<i>nas-cg</i>
0	0.00277	0.00172	0.00127	0.10782
1	0.00277	0.00507	0.03580	0.00945
2	0.00230	0.00507	1.65059	0.02015
3	0.00230	0.00507	0.03534	0.00694
4	0.00034	0.00514	0.41795	0.00124
5	0.00034	0.00514	0.03542	0.00124
11	0.00232	0.00505	0.00443	0.00019
13	0.00034	0.00510	0.00443	0.00017
21	0.00034	0.00510	0.00068	0.00017

**Table 7.13:** NAS benchmarks, average poll point intervals on x86.  
(times in milliseconds)

The results of state capture and recovery experiments performed on the NAS benchmarks are presented in Table 7.14. These results are similar to those observed for the applications discussed in Section 7.3. Again, we find that state capture and recovery costs are primarily a function of state size, and that state recovery across incompatible data formats introduces significant overhead. An interesting point of note in these experiments is the wide range of state sizes. Each of the programs is computationally demanding, but their state sizes range from hundreds of bytes to tens of megabytes.

	<i>nas-is</i>	<i>nas-ep</i>	<i>nas-mg</i>	<i>nas-cg</i>
<i>Chckpt size</i>	16785727	314	7025076	960511
$t_{chckpt}$	535.34	0.76	550.67	22.57
$t_{restart}$	1054.31	0.60	516.67	63.97
$t_{restart, \alpha}$	1055.67	0.63	523.03	63.87
$t_{restart, rs/6000}$	1529.84	0.61	602.51	84.98
$t_{restart, mips}$	1534.34	0.61	601.02	86.07
$t_{restart, sparc}$	1569.59	0.61	596.77	84.75

**Table 7.14:** NAS benchmarks, time to checkpoint/restart on x86.  
(sizes in bytes, times in milliseconds)

## 7.5 Environmental Simulation

Although the programs discussed thus far resemble components commonly found in scientific codes, they do not capture the full complexity of a complete application. Production scientific applications are typically much larger than the examples examined thus far, they rely on compilation features such as separately built modules, they make use of more complex data structures, and they utilize run-time features such as file I/O. To demonstrate the ability of the Process Introspection system to handle full-scale applications, and to examine the performance of the system in this more demanding context, we applied the system to real, non-trivial scientific programs in production use. In this section and Section 7.6 we describe our experiences and performance results obtained with these applications.

The first full-scale application to which we applied the Process Introspection system was a program implementing a global carbon productivity and geographic plant distribution model described by Woodward, Smith, and Emanuel [94]. This program, *lai*, computes the leaf area index for a specified set of geographic locations. The leaf area index for a given region is defined to be the amount of leaf surface area per ground surface area over that region. This metric provides a measure of the geographic distribution of plant life, which can in turn be included in climate mod-

eling simulations. Since vegetation has a significant effect on the environmental systems that control the climate, the computation of the geographic distribution of plant life is an important part of predicting future climactic changes due to influences such as fossil fuel emissions.

The model employed by *lai* simulates plant systems on various levels of resolution and in various time scales based on input climactic data for the region in question. At the lowest level of resolution, the cell-level biochemical processes associated with photosynthesis are simulated. Higher levels simulate leaves, complete plants, and finally the entire vegetation canopy. Input data for each simulated region includes temperature, day length, wind, precipitation, and soil chemistry.

In terms of program structure, *lai* is significantly more complex than the examples presented in Section 7.3 and Section 7.4. The program is approximately 4000 lines of C code, and relies on separate compilation of 22 source files. In addition to this larger size, *lai* makes use of more complex data structures than the previous examples, and relies on file I/O for input of climactic data and output of simulation results.

We ran performance experiments on *lai* using a sample data set of seventeen 2500 km<sup>2</sup> geographic regions. As in the case of the applications discussed thus far, we first examined introduced overhead and compiler effectiveness. These results are summarized in Table 7.15. We found that the performance overhead introduced by Process Introspection into *lai* is significant. Although the optimizer effectiveness for this application was high, all of the poll-point placement policies applied resulted in very frequent placements, and as a result we measured  $O_{opt} > 10\%$  in all cases.

Policy	$O_{norm}$	$E$	$O_{opt}$
0	13.4%	1.00	13.0%
1	12.9%	0.98	15.0%
2	13.2%	0.99	14.3%
3	12.9%	0.99	14.6%
4	12.9%	0.98	15.4%
5	12.6%	0.97	15.8%
11	12.8%	0.97	16.6%
13	13.3%	0.96	18.3%
21	13.1%	0.96	18.2%

**Table 7.15:** LAI, overhead and optimizer effectiveness.

We have already observed that relatively conservative policies generally lead to good performance. In this case, it appears that all of the implemented placement policies are too aggressive. It might be observed that one obvious way to achieve potentially more conservative placement would be to apply policy 0 (the mandatory-only placement policy), but with the addition of the mandatory poll point elimination optimization described in Section 3.3.3. To investigate this possibility, we implemented this new policy in APrIL, numbering the policy -1 (as it can certainly result in no more placements than policy 0). Unfortunately, when we applied this policy to *lai* we found that it resulted in the same poll point placement decisions as made by policy 1. The loop structure of this application results in no optional poll point placements being performed by policy 1, which reduces policy 1 to mandatory placements only plus the mandatory poll point elimination optimization—i.e. policy -1. Thus, our attempt to improve the performance of *lai* by implementing a more conservative poll point policy was not successful.

In *lai*, we have an application for which it appears that the current implementation results in a level of performance overhead that, while not terrible, is slightly greater than would generally be considered acceptable. This leads naturally to the question: why? What attributes of the application and/or the current compiler implementation result in high overhead, and where (if anywhere) is there room for improvement? An examination of the *lai* source code structure with respect to the implemented placement strategies sheds light on the problem. The code for this application is structured to make use of many small functions, the definitions of which are spread among a number of different source files. Furthermore, the APrIL mandatory poll point placement elimination optimization must necessarily be conservative—if a function is not defined in the current source file being translated, APrIL *must* assume that the function can initiate state capture operations, and thus mandatory poll point placements at calls to the function are required to preserve correctness. This combination of factors results in unnecessary poll point placements at the call sites of simple functions that will never initiate state capture operations. In *lai*, a large number of these unneces-

sary placements are performed. This leads to a general observation: a program divided into many fine-grained subroutines and defined in many separately compiled source files can result in relatively high overhead under the current implementation.

This observation initiates a further question—can this bad interaction be avoided? One immediate possibility is obvious: the programmer could restructure the code to use larger subroutines, or could combine the subroutines into fewer source files. Either restructuring strategy avoids the undesirable interaction with the APrIL poll point placement heuristics. However, this solution is undesirable since it defeats our goal of automation—the programmer should not be required to structure applications differently to utilize a state capture and recovery mechanism effectively. A better solution would be to incorporate the notion of a “project” into APrIL—a collection of separately compiled but related program modules. For each program module, APrIL could store information about the poll point counts in each contained function (for example, APrIL might write out an auxiliary file listing this information for each source module). This would allow the mandatory poll point elimination optimization described in Section 3.3.3 to be applied across source modules and could effectively reduce the performance overhead observed for this application. This extension to APrIL is the subject of future work on the system. Note, separate compilation alone was not sufficient to cause increased overhead—usage of many small subroutines was also necessary. In Section 7.6 we examine full-scale applications that utilize separate compilation but that achieve low introduced overhead comparable to that observed for the smaller applications discussed in Section 7.3 and Section 7.4.

Given the interaction between this application and our poll point placement heuristics, we expect that the observed poll point counts for *lai* would be high, and thus the average poll point intervals would be very small. In Table 7.16 we present the poll point counts and sample poll point intervals for *lai*. The results confirm intuition—*lai* performs with very low poll point intervals (less than 10 microseconds) for all placement policies.

Policy	Poll Points	x86	sparc
0	77428159	0.000364	0.000587
1	73884216	0.000381	0.000650
2	75624427	0.000374	0.000600
3	75624427	0.000374	0.000603
4	75637639	0.000374	0.000626
5	75637639	0.000374	0.000632
11	75686477	0.000367	0.000651
13	75699689	0.000385	0.000644
21	75699689	0.000385	0.000649

**Table 7.16:** LAI, poll point counts and average poll point intervals.  
(times in milliseconds)

Our final set of experiments with *lai* measured the state capture and recovery times for this program. In Table 7.17 we present sample results. During execution, this application keeps in memory only the state associated with one data point. The state for other geographic locations (e.g. climactic data, output leaf area indices, etc.) are maintained in files. This attribute causes *lai* to have a relatively small state size, and thus state capture and recovery costs are very low.

	alpha	x86	rs/6000	mips	sparc
<i>Chckpt size</i>	6782 bytes				
$t_{chckpt}$	1.96	2.07	22.77	11.54	6.77
$t_{restart,alpha}$	2.93	3.64	8.03	12.43	8.53
$t_{restart,x86}$	2.94	3.66	8.01	12.34	8.71
$t_{restart,rs/6000}$	2.93	3.73	7.82	12.49	7.99
$t_{restart,mips}$	2.93	3.70	7.82	12.39	8.01
$t_{restart,sparc}$	2.93	3.68	7.86	12.12	8.28

**Table 7.17:** LAI, checkpoint/restart costs.  
(times in milliseconds)

## 7.6 Biological Sequence Comparison

The second set of full-scale applications examined for this performance study consisted of two biological sequence comparison programs. Using modern techniques, biochemists can deter-

mine the sequence of a protein significantly more easily than they can determine a protein's function. By comparing a sequence to other proteins with known functions, biochemists can gain insight into the possible function of the protein. Comparison of sequences is essentially a string distance problem where DNA and protein molecules are represented by strings of nucleotides and amino acids, respectively. A typical query involves comparing a new sequence with unknown function to a library of published sequences. For each comparison a score is generated. When the full library comparison is complete, the results are presented in non-increasing order.

For this performance study, we examined two sequence comparison programs: *sw* (smith-waterman [78]), and *fa* (fasta) [66]. The *sw* version of the program performs a rigorous, quadratic linear programming algorithm. The *fa* version employs a heuristic that maintains a lookup table containing regions of high densities of identity resulting in execution speeds between 20 and 100 times as fast as the rigorous *sw* version. The *sw* version is approximately 6500 lines of C code, and the *fa* version is approximately 9000 lines. Each version is divided into 14 separately compiled modules, 12 of which are shared between the versions as they implement functionality that is common to both. The programs utilize file I/O to access a protein library and query sequence. Both versions make use of complex nested data structures, pointers (including function pointers), and other advanced language features that must be handled correctly by APRIL.

We ran our standard set of performance experiments on *sw* using a sample library containing 500 sequences. For the faster *fa* version, we increased the library size to 3000 sequences. As for previous applications, our initial experiments characterize the introduced overhead and optimizer effectiveness. The results for *fa* are listed in Table 7.18, and the results for *sw* are listed in Table 7.19. The results for *fa* again confirm our observation that conservative poll point placement is generally most effective. For this application, policies 0 and 1 result in acceptable levels of net overhead, whereas more aggressive policies both impede optimizations and introduce significant additional overhead, a combination that leads with certainty to high net overhead. The results for

*sw* are somewhat less sensitive to placement aggressiveness, although the most aggressive placement policies seriously degrade performance.

Policy	$O_{norm}$	$E$	$O_{opt}$
0	3.1%	0.94	9.5%
1	4.1%	0.96	9.0%
2	13.1%	0.90	26.9%
3	17.1%	0.93	26.3%
4	16.0%	0.89	31.7%
5	14.7%	0.86	34.4%
11	14.4%	0.88	30.9%
13	17.9%	0.88	34.9%
21	19.1%	0.89	34.7%

**Table 7.18:** FASTA, overhead and optimizer effectiveness.

Policy	$O_{norm}$	$E$	$O_{opt}$
0	1.2%	0.96	5.1%
1	1.6%	0.95	7.4%
2	1.2%	0.94	7.5%
3	1.3%	0.94	7.8%
4	1.8%	0.94	8.5%
5	2.5%	0.95	7.9%
11	9.2%	0.87	27.3%
13	9.8%	0.86	30.9%
21	10.2%	0.88	28.8%

**Table 7.19:** Smith-Waterman, overhead and optimizer effectiveness.

As always, introduced overhead is only half of the story. Without small average poll point intervals, low overhead is meaningless. In Table 7.20 and Table 7.21 we list the poll point counts and sample average poll point intervals for *fa* and *sw*, respectively. In general, most poll point placement policies achieve small poll point intervals, with the exception of the most conservative policies applied to *sw*. Again, we note the result that for both applications, relatively conservative poll point policies exist that provide introduced overhead below 10% and very low poll point intervals. For example, policy 1 achieves good performance with *fa*, whereas a slightly more aggressive policy such as 2 or 3 leads to better performance for *sw*.



Policy	Poll Points	x86	sparc
0	266559	0.006671	0.020547
1	1360568	0.001327	0.003895
2	10230798	0.000181	0.000590
3	10366695	0.000178	0.000554
4	11091474	0.000171	0.000536
5	11218210	0.000168	0.000557
11	11973635	0.000156	0.000494
13	12825150	0.000148	0.000486
21	13365830	0.000143	0.000463

**Table 7.20:** FASTA, poll point counts and average poll point intervals.  
(times in milliseconds)

Policy	Poll Points	x86	sparc
0	212198	0.108863	0.271430
1	378530	0.063050	0.152779
2	692001	0.034477	0.083227
3	699047	0.034127	0.084283
4	1762417	0.013545	0.033745
5	1769429	0.013487	0.033298
11	103436444	0.000241	0.000629
13	104506826	0.000237	0.000642
21	104531290	0.000238	0.000626

**Table 7.21:** Smith-Waterman, poll point counts and average poll point intervals.  
(times in milliseconds)

Our final set of experiments measured the performance of state capture for these applications. In Table 7.22 and Table 7.23 we present the state capture and recovery costs for *fa* and *sw*, respectively. We note in these experiments that the state capture and recovery costs are somewhat high, especially for the *fa* application which has a larger state size, and especially on the slower test platforms. This is not entirely surprising given that our prototype implementation of the PIL is not highly optimized, and these applications have very complex states. However, it would be desirable to achieve better performance than this, and the current implementation provides a mechanism for improving state capture and recovery costs.

	alpha	x86	rs/6000	mips	sparc
<i>Chckpt size</i>	2801497	2761417	2761417	2761417	2761417
$t_{chckpt}$	416.75	777.18	3251.88	4210.58	3197.86
$t_{restart,alpha}$	621.38	847.75	4393.22	6506.65	3317.85
$t_{restart,x86}$	619.59	826.49	2537.36	3742.57	2929.55
$t_{restart,rs/6000}$	675.25	859.25	2456.70	3783.78	2727.21
$t_{restart,mips}$	671.57	853.96	2439.23	3619.18	2762.01
$t_{restart,sparc}$	674.99	853.77	2448.74	3649.23	2651.87

**Table 7.22:** FASTA, time to checkpoint/restart.  
(sizes in bytes, times in milliseconds)

	alpha	x86	rs/6000	mips	sparc
<i>Chckpt size</i>	1197083	1157003	1157003	1157003	1157003
$t_{chckpt}$	162.60	283.02	1085.98	1529.52	916.34
$t_{restart,alpha}$	200.13	331.16	2792.45	5490.56	2449.60
$t_{restart,x86}$	204.59	320.06	1001.76	1893.06	1047.10
$t_{restart,rs/6000}$	239.56	331.24	959.66	1789.38	1002.62
$t_{restart,mips}$	243.59	328.83	954.17	1728.45	947.75
$t_{restart,sparc}$	242.13	329.58	969.83	1686.68	972.11

**Table 7.23:** Smith-Waterman, time to checkpoint/restart.  
(sizes in bytes, times in milliseconds)

In Section 5.3.3 we described a PIL mechanism that allowed user-coded marshalling functions to be registered for use with structured data types in place of the automatic mechanism based on the type description table maintained by the PIL. Furthermore, in Section 6.2.6 we described how APrIL supports the automatic generation of these marshalling functions. Thus far we have reported performance for programs transformed without this APrIL compile-time option enabled. To investigate the effect of APrIL-generated marshalling functions, we ran state capture and recovery experiments with *fa* and *sw* transformed with and without marshalling functions. The results of these experiments are presented in Table 7.24.

We found that a 12%-15% reduction in state capture and recovery costs was possible using this compile time option. The cost of this reduction was a growth in the executable size by approximately 15%, although some of this growth could be avoided with an improved implementation.

Again, due to separate compilation, APrIL generates redundant marshalling functions. With an improved “project” style grouping facility, APrIL could avoid redundant marshalling function generation and could achieve equal performance gains with significantly less code growth.

	FASTA		Smith-Waterman	
	<i>without</i>	<i>with</i>	<i>without</i>	<i>with</i>
$t_{chckpt}$	777.18	655.87	283.02	236.86
$t_{restart, alpha}$	847.75	735.70	331.16	286.18
$t_{restart, x86}$	826.49	715.05	320.06	272.31
$t_{restart, rs/6000}$	859.25	745.00	331.24	284.08
$t_{restart, mips}$	853.96	736.60	328.83	281.43
$t_{restart, sparc}$	853.77	736.65	329.58	283.51

**Table 7.24:** Performance with and without marshalling functions.  
(times in milliseconds)

## 7.7 Performance Discussion

In this chapter we have examined the details of a performance study of the Process Introspection system. At this point, it is worth summarizing the main results of this study.

Perhaps the single most important result of our experiments is that Process Introspection, as described in this dissertation, and as realized in our prototype implementation, can achieve low introduced overhead and small average poll point intervals when applied automatically by the APrIL compiler. This result was achieved using straightforward poll point placement heuristics that are simple to implement and that can be applied efficiently. It should be highlighted that this result was not an obvious one given the general design of Process Introspection as described in Chapter 3. Poll point placement clearly results in the fundamental performance trade-off between introduced overhead and state capture request wait time, both of which we desire to be as small as possible. But does any point of acceptable trade off between the two exist for real programs? Furthermore, if acceptable placements that result in reasonable performance in both dimensions do exist, can these placements be performed automatically by a compiler? The performance study described in this chapter answers both of these questions in the affirmative. Process Introspection

can be applied both automatically and efficiently.

Beyond this most fundamental result, we have characterized a number of performance attributes of Process Introspection in general, and of the current system implementation. First, we have learned that conservative poll point placement strategies perform well in general. Relatively sparse placements almost always lead to low introduced overhead, and rarely result in long on-average poll point intervals. This result does not imply that the placement of optional poll points is not valuable. In almost all of the test application sets we found cases where using only mandatory poll points resulted in very long poll point intervals. This result is not surprising—it is not uncommon for scientific programs to execute long-running loops with no subroutine invocations. Given this behavior, optional poll point placement is almost certainly desirable. Given this, we found that placement of optional poll points in only a few large outer loops was best. Again, this result is not surprising given the known possibility that Process Introspection can hinder optimizations as discussed in Section 4.3.

Not all of the results of the study were positive. In Section 7.5 we discussed a limitation of the system to handle certain program structures efficiently. For the application discussed in this section, it appears that the program structure puts a lower bound on the number of poll points that the current APrIL implementation must place that is greater than the number of poll points that would lead to a reasonable level of performance/overhead trade-off. To address this problem, we plan to improve the APrIL mandatory poll point elimination optimization to operate in a cross-module mode. We hypothesize that this enhancement will improve APrIL's ability to handle applications with the described problematic structure.

# Chapter 8

## Extensions

Although the Process Introspection model described in Chapter 3 is very general, and could be applied to a wide range of programming languages, the implementation of the system presented in Chapter 5 and Chapter 6 is specific to single-threaded ANSI C programs. In this chapter we examine the implementation issues involved in extending the system to a wider range of programming environments. In Section 8.1 we discuss the extension of the system to handle additional programming languages, and in Section 8.2 we examine the issues involved in supporting multi-threaded execution.

### 8.1 Additional Programming Languages

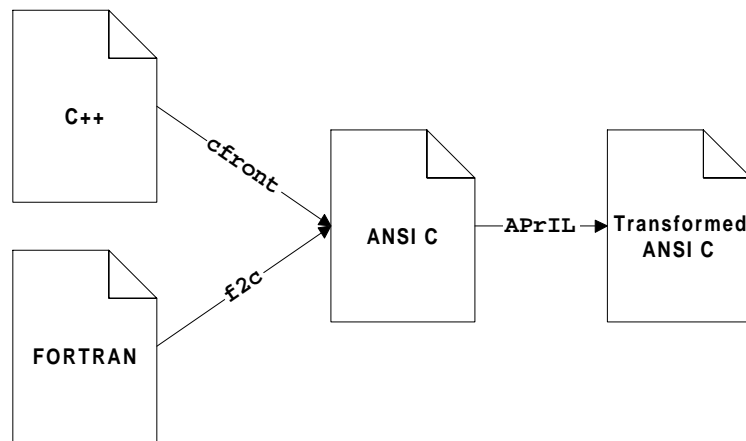
The current implementation of the Process Introspection system is applicable to programs written in only one programming language: ANSI C. In practice, we would like a system to support programs written in a variety of programming languages. In this section, we examine the issues involved in supporting an additional procedural language, Fortran 77<sup>1</sup>, and a language with object-oriented features, C++. Within the spectrum of programming languages, Fortran and C++ are relatively similar to the currently implemented language, C. However, examining the issues involved in Process Introspection implementations for these relatively simple cases provides insight into the larger issue of multi-language support. Furthermore, the vast majority of language usage in high-performance computing is restricted to C, C++, and Fortran, and thus examining support for these languages is valuable from a practical standpoint.

A first, most obvious approach to supporting additional languages is to retain the existing system implementation without modification, and translate programs written in other programming

---

1. Subsequently, we refer to Fortran 77 simply as Fortran. Support for the additional features introduced in Fortran 90, vendor specific Fortran implementations, and other later Fortran dialects are not covered.

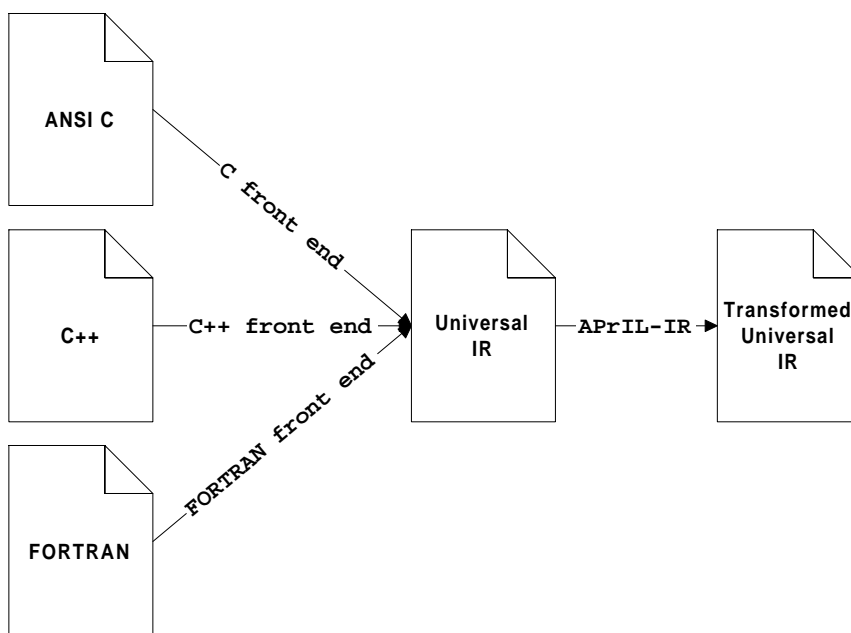
languages into C. This design is depicted in Figure 8.2. C is a fully general programming language, and could be used as a target for front ends for a variety of other languages. In particular, some form of support for translation to C exists for both C++ and Fortran, in addition to other languages such as Java and Pascal. This strategy leads to a number of potential problems. First, the C code generated by alternate language front ends may not be acceptable input for APrIL. For example, **f2c**, the de-facto standard Fortran to C translator [27], makes use of union data structures (which are not currently supported by APrIL) in some of its transformations. Furthermore, the translated code may make use of libraries that would also need to be modified to include state capture and recovery capabilities. For example, **f2c** relies on a library of functions implementing the intrinsic operations provided by Fortran. Some of these, for example the I/O operations, would need to be modified to interoperate with the PIL. Finally, the performance of the transformed code may be somewhat degraded. For example, information needed to perform certain optimizations on Fortran code may be lost in a C translation. Despite these drawbacks, this approach is attractive for its simplicity, and has been demonstrated using simple Fortran codes transformed by **f2c**.



**Figure 8.1:** Alternate language support using the existing implementation.

An alternative to using the existing implementation which is based on C as an intermediate representation would be to employ an alternative intermediate representation. All supported lan-

guages would be transformed to this universal IR, which would be transformed using the Process Introspection model described in Chapter 3. This design is depicted in Figure 8.2. Although it is attractive for its generality, this scheme makes the least use of existing tools, requiring front ends for all supported languages, as well as a back-end optimizing compiler for the IR on all target platforms.

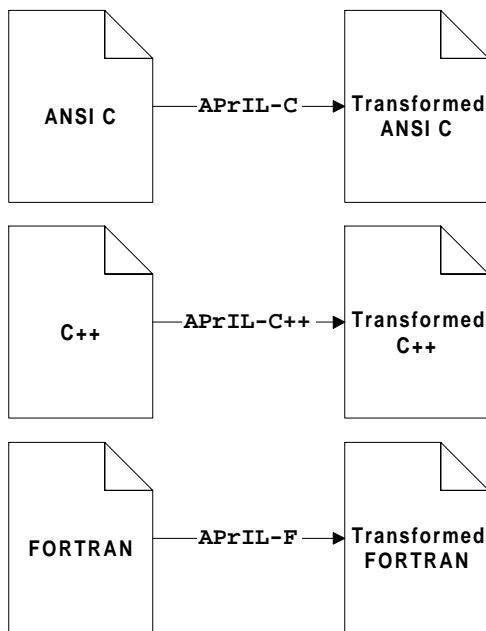


**Figure 8.2:** Alternate language support using a universal IR.

A third design alternative, and the one which we will explore in greater depth in the remainder of this section, is to use source-to-source transformation techniques for all supported languages analogous to those currently employed by APrIL for C, as depicted in Figure 8.3. This third approach requires a version of APrIL specialized for each supported language, but avoids some of the negative performance impact of the first design alternative, and utilizes existing back-end compiler technology unlike the second.

Assuming direct source-to-source transformation is employed, we must customize the APrIL transformations to utilize the features available in each input language. Although the transformations will correspond generally to those employed for C, the details must take into account various

language constructs and features. In Section 8.1.1 and Section 8.1.2 we examine the implementation of the APrIL transformations for Fortran and C++, respectively.



**Figure 8.3:** Alternate language support using source-to-source translation.

### 8.1.1 Fortran

In most respects, the source-to-source transformations that must be applied to Fortran by an APrIL compiler are only syntactically different from their analogues in C. For example, in Figure 8.4 we depict an optional poll point in Fortran, and in Figure 8.5 we depict a mandatory poll point<sup>1</sup>. These correspond exactly to their C counterparts, except that they are expressed using Fortran syntax.

---

```

10 CONTINUE
   IF (PLSTAT .EQ. PLCHCK) THEN           Check for checkpoint request.
      CALL PLPUCL(10)                     Remember the code location.
      PLSTAT = PLCINP                     Mark checkpoint in progress.
      GO TO 100                            Jump to the subroutine epilogue.
   END IF

```

---

**Figure 8.4:** An optional poll point in Fortran.

1. We assume a Fortran interface to the PIL. Identifiers in this interface begin with the prefix **PL**.



---

20 CONTINUE	
CALL PROC	<i>Subroutine call requires a poll point.</i>
21 CONTINUE	
IF (PLSTAT .EQ. PLCINP) THEN	<i>Check for checkpoint in progress.</i>
CALL PLPUCL(20)	<i>Save restore location before call.</i>
GO TO 100	<i>Jump to the subroutine epilogue.</i>
ELSE IF (PLSTAT .EQ. PLCHCK) THEN	<i>Check for checkpoint request.</i>
CALL PLPUCL(21)	<i>Save restore location after call.</i>
PLSTAT = PLCINP	<i>Mark checkpoint in progress.</i>
GO TO 100	<i>Jump to the subroutine epilogue.</i>
END IF	

---

**Figure 8.5:** A mandatory poll point in Fortran.

In some cases, the transformations must take into account differences in the languages' basic mechanisms. For example, C parameters are passed by value, but Fortran parameters are passed by reference. This difference in language features affects the implementation of function prologues and epilogues. Given the call-by-value parameter passing semantics of C, the function prologues and epilogues in APrIL-transformed C code are responsible for restoring and saving actual parameter values. In Fortran, function and subroutine prologues and epilogues can safely ignore parameters, since they will be restored and saved in calling function or subroutine invocations. For example, Figure 8.6 depicts a Fortran subroutine prologue transformation. Note that when the local subroutine activation state is recovered, the values of parameters are not read from the checkpoint. This minor modification aside, the prologue transformation (and similarly, the epilogue transformation) is only syntactically different from its C counterpart.

In addition to the fundamental APrIL transformations, a Fortran version of APrIL must also perform similar pre-processing transformations to ensure that function calls appear only in simple expression statements, and that each expression statement contains only one function call. For example, a function call might appear in an expression that is a parameter to a write statement. This function call would need to be moved above the write statement, its result would be assigned to a temporary variable, and this temporary variable would be used in place of the function call in

---

```

SUBROUTINE SUM(X, TOTAL)
INTEGER X(100)
INTEGER TOTAL, I

```

---

(a) Original subroutine heading

---

```

SUBROUTINE SUM(X, TOTAL)
INTEGER X(100)
INTEGER TOTAL, I
INTEGER PLLOC

```

*Location to jump to on restore.*

```

IF (PLSTAT .EQ. PLREST) THEN
  CALL PLSRIN(I)
  CALL PLPOCL(PLLOC)
  IF (PLLOC .EQ. 20) THEN
    GO TO 20
  ELSE IF (PLLOC .EQ. 21) THEN
    CALL PLDONR
    GO TO 21
  END IF
END IF

```

*Check for restart in progress.  
Recover the value of local I.  
Recover the restore location.  
Jump to appropriate code location.  
Mark restart as done.  
Jump to appropriate code location.*

---

(b) The transformed subroutine heading

**Figure 8.6:** A function prologue transformation in Fortran.

the write statement. Similarly, if an assignment statement contained two function calls on its right hand side, the assignment would need to be split into two statements, one calling the first function and assigning its result to a temporary variable, and a second using the temporary variable in place of the moved function call. The implementation of these pre-processing transformations would be essentially the same as those for C, as described in Section 6.3.

Some language features in Fortran require additional transformations to be performed by APRIL. A good example of this is the Fortran **ENTRY** statement, which provides alternate entry points into a function or subroutine. Since calls to alternate entry points are essentially the same as normal subroutine invocations, these points would need to be marked as mandatory poll points. Furthermore, since these entry points could be called in the process of recovering the stack, each

entry point in a subroutine would need to be followed by a copy of the subroutine prologue (note, this is safe due to the restriction that **ENTRY** statement appear outside any **DO** loops or **IF** blocks). As with standard prologues, these prologues would not be executed during normal program flow.

An additional transformation requirement is introduced by the Fortran **ASSIGN** statement, which allows the assignment of a code label (i.e. line number) to an integer variable. The first requirement that this introduces is that the addresses of all statements, the locations of which are assigned to variables, be registered in the PIL code location table. This allows code locations to be described using the standard PIL pointer description mechanism. However, registering code locations is only half of the required solution. Since code locations are assigned to normal integer variables, we must introduce a mechanism for determining when variables contain code locations, and when they should be treated as normal integers. A simple solution that addresses this issue can be based on maintaining a list of code-location-containing variables. After each **ASSIGN** statement in the code, the target variable is added to the list. When variables that might ever be the target of an **ASSIGN** statement are the target of a normal assignment, they are removed from this list. Thus, at any point in execution, this code-location-containing variable list will have an up-to-date record of all integer variables that actually contain code addresses instead of normal integer values. At checkpoint time, a list of logical pointers to the variables on the list is saved, and the values of these variables are saved as logical pointer descriptions. During recovery, if a variable is found on the code location list, it is restored using pointer resolution.

A further issue that must be addressed in the source-to-source transformation of Fortran programs is the use of **COMMON** blocks. Variables included in common blocks are similar to global variables in C, and thus can be registered with the PIL using an analogous technique to that used for C globals. Recall, for transformed C code, an initialization routine is introduced which is executed before the main program begins, and which registers the locations and sizes of global blocks. As depicted in Figure 8.7, we can use a similar technique for Fortran. An initialization routine con-

taining **COMMON** statements to make all common blocks available to it. Inside the routine, variables within the common block are registered using an interface similar to the PIL interface introduced in Section 5.3.4.

However, use of common blocks and the Fortran **EQUIVALENCE** statement introduces potential problems for Process Introspection. Like the C union construct, these mechanisms allow variables, possibly of different types, to occupy the same regions of memory. For example, to save space, an **EQUIVALENCE** statement might indicate that a real array and an integer array overlap in memory. As long as the arrays are not needed at the same time during the execution of the program, this type of memory optimization is safe. However, the overlap of memory regions of different types is problematic for the PIL. For example, if a memory region in a common block were registered with the PIL as containing integers, but during a state capture operation actually contained real values, a restart of the process on a different platform might restore the memory region incorrectly. One possible way to address this issue would be to disallow the overlay of incompatible types. If complete source were available, static checking could be performed by APrIL to ensure that only like data types were overlaid. If separate compilation were required, run-time checks could indicate mismatched memory overlays.

A final issue that must be addressed is I/O. Unlike in C, I/O is part of the Fortran language. In the current APrIL implementation, the issue of I/O is not addressed by the compiler, but is instead handled automatically by wrapper library routines. In Fortran, this approach is not possible, and some support is required from the APrIL compiler. This support would involve transforming **READ** and **WRITE** statements into calls into a PIL-interoperable I/O library, much like that described in Section 5.4.

---

```

SUBROUTINE ONE
DIMENSION X(10)
COMMON /A/ X
    . . .

SUBROUTINE TWO
COMMON /B/ I, J
    . . .

SUBROUTINE PLINIT      Initialization event handler.
DIMENSION X(10)
COMMON /A/ X           Make all common blocks available.
COMMON /B/ I, J
CALL PLGREL(X, 10)    Register an array of reals.
CALL PLGINT(I, 1)     Register a scalar integer.
CALL PLGINT(J, 1)     Register a scalar integer.

```

---

**Figure 8.7:** Registration of variables in common blocks.

### 8.1.2 C++

Since C++ is a superset of C, the set of transformations that would be required to support source-to-source Process Introspection compilation for C++ is a superset of those performed by APrIL as described in Chapter 6. The main difficulty in extending the existing set of transformations to apply to C++ programs is the complex set of language features provided by C++, and the interactions between these features.

The primary difference between C and C++ is the introduction of classes in C++. In some respects, classes do not affect the way APrIL works. Assuming data hiding directives (i.e. **private** and **protected** specifiers) are enforced and then removed, class instances can be treated much like C **struct** variables. Furthermore, member functions can be transformed in the same way as static functions, and their invocations can be transformed into mandatory poll points in the standard way. The primary difficulties introduced by classes involve constructors, destructors, and

the **new** operator.

Class constructors and destructors are special methods whose invocation is guaranteed at certain specific times during program execution. For example, a constructor is invoked on an object when that object enters scope (i.e. when execution reaches the object's point of declaration) or is dynamically allocated from the heap. Conversely, destructors are automatically invoked on objects when they leave scope or are returned to the heap. These semantic guarantees have problematic interactions with the APrIL transformations as implemented. First, in order to utilize function epilogues, APrIL moves all variable declarations to the beginning of a function. This transformation could alter the point in control flow at which constructors were invoked, and could thus change the meaning of the program. Furthermore, the guarantee of automatic constructor invocation would interact incorrectly with the state recovery operation. Constructors are normal functions, and would be transformed to include poll points and so on (e.g., it would be fair to assume that a checkpoint might be initiated during the execution of a long-running constructor). However, since stack reconstruction relies on the exact functions that were active during checkpoint construction being invoked in the appropriate order, if constructors were automatically invoked as stack frames were entered, the stack could not be correctly recovered. Destructors cause a similar problematic interaction with the stack capture operation. Destructors can initiate or participate in stack save operations, and thus if they were automatically invoked as functions returned during the stack capture, an incorrect representation of the stack would be produced.

A simple solution to the problems introduced by constructors and destructors can be based on the transformation of these automatically invoked methods into normal, explicitly invoked methods, as depicted in Figure 8.8. As a natural compliment to this transformation, explicit invocations of the introduced methods would need to be generated in all locations where the automatic methods would have been executed. For example, if an object instance were declared in a given scope, the appropriate explicit constructor invocation would need to be placed at the entry to the scope,

and explicit calls to the destructor would need to be placed at all exits from the scope (e.g. a **break** statement from a loop, etc.). Also, explicit calls to constructors would need to be placed after each **new** operation, and before each **delete** operation.

<pre> class ex {     int data; public:     ex(int);     ~ex(); };  void f() {     ex e(1);     . . . } </pre>	<pre> class ex {     int data; public:     void ctor(int);     void dtor(); };  void f() {     ex e;     e.ctor(1);     . . .     e.dtor(); } </pre>
(a) Before transformation	(b) After transformation

**Figure 8.8:** C++ constructor/destructor transformations.

A further complexity introduced by classes in C++ is dynamic memory allocation operations. Unlike C, in which an untyped library interface is used for memory allocation (e.g. **malloc**), C++ provides type-specific, language-level memory allocation operators (i.e. **new** operators). Since the C++ **new** operator not only allocates memory, but also initializes data internal to allocated objects (e.g. virtual function lookup tables for dynamic binding), it cannot be trivially replaced by calls to C heap allocation library routines. In conflict with this, the existing PIL implementation only provides a wrapper interface for standard C memory allocation routines. Furthermore, the automatic capture and recovery of heap-allocated memory blocks by the PIL is predicated on the use of these routines, and pointer description and resolution relies on the list of heap allocated memory block descriptions maintained by these routines.

A solution that could bridge the gap between the required use of the **new** operator by C++

programs, and the required use of wrapper routines by the PIL could be based on the generation of **new** wrapper routines by APRIL—introduced functions to return the result of a call to the **new** operator. The addresses of these routines could be registered with the PIL, and when allocation of a memory region of a given type was required, a call to a PIL allocation routine could be used. This routine could maintain a list of memory block descriptions in the same way that **PIL\_Malloc** does (see Section 5.3.5), but could call the registered **new** wrapper routines to perform actual memory allocation instead of calling on **malloc**. Furthermore, the wrapper routines could be used internally by the PIL heap allocation module at restart time when all heap-allocated memory blocks must be re-allocated. Again, calls to the appropriate **new** wrapper could be used in place of calls to **malloc**. An example of the transformations associated with this scheme is depicted in Figure 8.9. Note, the determination of the type allocated is greatly simplified by C++ as compared to the heuristic approach described in Section 6.2.5 for C. The allocated type is always the class specified to the **new** operator.

---

```

class ex2 : public ex1 {
    . . .
};

void f() {
    ex1 *e = new ex2;
}

```

---

(a) Original code

---

```

int PIL_ex2;
class ex2 : public ex1 { . . . };

ex2 *new_ex2() {
    return new ex2;
}

PIL_Init() {
    PIL_ex2 = PIL_RegisterClass();
    PIL_RegisterNew(PIL_ex2, new_ex2);
}

void f() {
    ex1 *e = PIL_New(PIL_ex2, 1);
}

```

---

(b) Transformed code uses wrapped **new****Figure 8.9:** Wrapper **new** operator registration.



The use of templates in C++ introduces the need for additional APRIL transformations. Consider the example template function depicted in Figure 8.10. When the prologue and epilogue are

---

```

template <class T>
T vectorSum(T *x, int n) {
    T ret = 0;
    int i;
    for(i=0;i<n;i++) ret+=x[i];
    return ret;
}

```

---

**Figure 8.10:** A template function.

generated for this function, explicit PIL typed I/O invocations to recover and save the value of the local variable `ret` must be included. As described in Section 5.3.3, the PIL typed I/O interface requires the PIL type description number of the type in question to be provided. In a template environment, this type number cannot be specified by referring to a specific global variable as is typically done in APRIL generated code, since the type in question will vary with different template instantiations. A solution to this problem can be based on the C++ function overloading mechanism. Overloaded functions can be generated to accept a dummy argument of a certain type, and return the type number for that type, as depicted in Figure 8.11. In place of the type number for the local variable in question, a call to the overloaded type number function is performed, thus allowing the templated code to obtain the correct type number in all instantiations. Note, as depicted in Figure 8.11, these functions can be inlined and thus would add no run-time overhead compared to the explicit use of global variables containing type numbers.

---

```

inline int PIL_GetTypeNum(ex1 x) { return PIL_ex1; }
inline int PIL_GetTypeNum(ex2 x) { return PIL_ex2; }

```

---

**Figure 8.11:** Overloaded type number functions.

Templates lead to a further possible complication for APRIL transformations. Note, in Figure

8.10 the operator “+=” of the template parameter class is invoked. In some cases, this might be a user-defined overloaded operator. In other cases, this might be a pre-defined operator of a basic C++ type. Outside of a templated scope, use of overloaded operators is an easily detectable alternative syntax for function invocation. As such, it can be treated by APrIL in the same way that function invocation is, providing targets for mandatory poll points, and affecting the operation of the pre-processor. However, in a templated scope, APrIL cannot discern between basic operators and user-defined operators. To be safe, APrIL must assume that all operators invoked on template parameter classes are overloaded operators. In the worst case, this assumption may lead to unnecessary mandatory poll point placement, and thus decreased performance, but in all cases will preserve the correctness of the code with respect to the Process Introspection transformations.

A final issue that we will discuss is the C++ exception mechanism. At any time during execution, C++ code can throw an exception consisting of a normal C++ object. When an exception is thrown, function activations are terminated from the top of the activation stack down, until one is found that has declared itself ready to “catch” the exception. Catching the exception consists of executing a **catch** statement that accepts as a parameter an object of the type (or of a type derived from the type) that was thrown. Upon receiving the thrown object, the program can perform arbitrary exception handling actions. An example of C++ exception handling is depicted in Figure 8.12. The syntactic rules imposed by the C++ **catch** statement are problematic for the APrIL transformations. The primary difficulty is that the object that is caught must be declared within the **catch** statement, but C++ **goto** statements (used during stack recovery) cannot cross object declarations. Outside the context of exception handling code, we can address this issue by moving variable declarations to the beginning of the function, using explicit constructors and destructors when necessary as described above. In a catch statement, however, we cannot perform this transformation. The result of this limitation of C++ is that, although a complete stack capture could be initiated from within exception handling code (i.e. from within code executed in a **catch** state-

ment after an exception has been thrown), the stack could not be recovered to such a state using only the goto mechanism.

---

```

class ex {
public:
    int errno;
    ex(int e) { errno=e; }
};

void throwIt() {
    throw ex(1);
}

main(){
    try {
        throwIt();
    }
    catch (ex e) {
        . . .
    }
}

```

---

**Figure 8.12:** C++ exception handling example.

A simple solution to this exception handling problem could be to disallow the initiation of state capture operations from within exception handling code. An implementation of this restriction could be constructed by maintaining a global variable indicating when exception handling is in progress. When a **catch** block is entered, the variable would be set. When the block was exited, the variable would be cleared. At all poll points, besides checking for a checkpoint request (i.e. examining the value of the **PIL\_CheckptStatus** variable), the “exception in progress” variable could be checked. If exception handling were in progress, the state capture operation could be deferred. Note, this solution could cause long checkpoint request wait times in the presence of long-running exception handlers.

## 8.2 Supporting Threads

In Section 8.1 we considered design issues associated with supporting additional program-

ming languages. Another dimension in which the current Process Introspection implementation can be expanded is to support additional programming paradigms. In this section we examine the design of extensions to the Process Introspection Library to support one such additional paradigm: programming with multiple threads of control. Multi-threaded programming is an increasingly common approach to writing applications that can take advantage of multiprocessor workstations, and for expressing the possibility of computation and I/O overlap for increased performance.

A design to support multiple threads is a natural extension of the currently implemented single threaded model. In the same way that a single thread captures and restores its stack and execution environment using the native subroutine return and call mechanisms, multiple threads can concurrently capture their individual stacks. A checkpointing process in a multithreaded environment must record a description of the set of active threads at the time of state capture—for example, the number of threads that was running, the starting function for each thread, and the stack data associated with each thread. Furthermore, state capture and recovery must address the issue of synchronization mechanisms. For example, if a lock were in use at the time of state capture, the checkpoint would have to record the identity of the thread that held the lock (if any) and the list of threads that were blocked waiting for the lock (if any). At restart time, the status of these threads with respect to the lock would need to be recovered.

The design we will present assumes that a low-level library interface for programming with threads is provided. Based on this low-level library, we will construct a higher-level threads library that interoperates with the PIL to support the capture and recovery of multithreaded processes. For simplicity, we will assume a minimal low-level thread library interface, which is depicted in Figure 8.13. This interface supports simple thread control operations—creation, termination, identity, and synchronization based on counting semaphores. Clearly, a richer interface could be assumed, such as the POSIX threads specification, which includes additional synchronization constructs, mechanisms for controlling thread scheduling, and so on. However, the basic threads interface we

have assumed simplifies the description of a design to support state capture and recovery without sacrificing adequate coverage of the salient issues. Note, in our low-level threads interface description, we have begged questions such as whether the implementation will reside in user or kernel space (or some combination), and whether the implementation will support true concurrency on multiprocessors. These and other related issues are orthogonal to the design we will present.

---

```

int  thread_create(void (*start_func)()); Create new thread, return thread id.
int  thread_curr(); Return the id of calling thread.
void thread_exit(); Terminate the calling thread.

int  semaphore_create(int start_value); Create a semaphore, return id.
void semaphore_p(int sem_id); Semaphore decrement.
void semaphore_v(int sem_id); Semaphore increment.

```

---

**Figure 8.13:** A simple threads interface.

In our design, we will provide wrapper functions for each of the operations depicted in Figure 8.13—each wrapper routine will support the same interface as its lower-level counterpart, but will maintain extra state and interact with the PIL in order to support the capture and recovery of multiple threads of control.

The first issue we must address is the capture and recovery of a description of the set of active threads in the process. To achieve this, the wrapper threads library maintains a list of active threads. Each element on the list contains a unique integer thread identifier assigned by the wrapper `thread_create` operation, the integer identifier of the thread specified by the low-level library, and a pointer to the starting function of the thread. Two identifiers are recorded for the thread to provide invariant thread identity across checkpoints and restarts. When the process is restarted, the low-level `thread_create` will be used to resume all threads. When this occurs, restarted threads may be assigned new thread identifiers. The wrapper routines thus maintain a second, invariant wrapper-level identifier for each thread that is preserved at restart time.

Given this active threads list, we can now specify the operation of the thread control opera-

tions in the interface. The wrapper-level thread creation routine first calls the low-level **thread\_create** operation to start the new thread. A record is then added to the active thread list containing a unique identifier generated for the thread, the identifier assigned to the thread by the low-level **thread\_create**, and the specified starting function for the thread. The wrapper **thread\_create** returns the wrapper-level unique identifier it generated for the thread. The wrapper current thread identity (**thread\_curr**) operation must map between the low-level identity of the calling thread and its wrapper-level identity. To do so, it can call the low-level **thread\_curr** operation, find the record on the active thread list that contains the resulting identifier, and return the wrapper-level unique identifier found in that record. Using the same technique, the **thread\_exit** operation looks up the calling thread on the active threads list, and removes the located record.

To capture and recover the state of the active threads set, the wrapper library registers event handlers with the PIL to be executed at checkpoint and restart time, respectively. The checkpoint handler writes the active threads list to the checkpoint, saving the wrapper-level unique identifier and a logical pointer description of the starting location for each thread. The low level thread identifier need not be saved since it will be overwritten at restart time. Note, the checkpoint handler only captures a description of the thread set, not the actual thread stacks and execution states—the threads will capture their own stack states using the normal Process Introspection mechanism. The restart handler reads the active threads list from the checkpoint, recovering the wrapper-level unique identifier and starting function pointer for each thread on the list. To resume the actual execution of the threads, the handler iterates over the records in the list, calling the low-level **thread\_create** for each, and in each case saves the low-level thread identifier for the record. Again, the restart handler does not restore the thread stacks, but instead only starts the threads running again, after which they use the normal Process Introspection mechanism to recover their stack state.

As described above, the threads in this design are expected to capture and recover their stack state using the standard Process Introspection model introduced in Chapter 3. The construction or recovery of thread stack descriptions can proceed concurrently for multiple threads, but introduces new requirements into the PIL stack management implementation (described in Section 5.3.7). Recall, the PIL stack management interface provides routines for writing and reading stack data to and from in-memory PIL buffers. The possibility of multiple threads introduces the requirement that the stack management routines maintain different buffers for each thread, preventing data associated with different stacks from being interleaved. A simple solution to address this requirement is to maintain a list of thread identifier/stack buffer pairs. When any stack management routine is invoked, it can use the `thread_curr` routine to determine the identity of the calling thread, and can read or write the specified stack data from or to the appropriate buffer in the list. Furthermore, since the stack management module automatically transfers in-memory stack frame buffers to and from the checkpoint at state capture and recovery times, this module must also save and recover the association between stack buffers and threads. The thread identifier/stack buffer association list could also serve this purpose.

In addition to the basic issue of preserving the association between stack data and its owner thread, the possibility of multiple threads of control introduces the need for synchronization at checkpoint and restart time. For example, suppose that two threads are executing in a process when a checkpoint request is received. Assume that one thread quickly reaches a poll point while the second thread continues normal execution. If the first thread begins to produce a checkpoint while the second thread continues to alter the state of the process, an inconsistent checkpoint could be produced. A similar problem can occur at restart time. Suppose two threads are recovering their stack state. Assume also that one thread quickly reaches the top of its call stack and resumes normal execution, while the second thread continues to recover its stack. While the first thread is executing normally and modifying the process's state, the second thread may be recovering data that

overwrites (i.e. rolls back) the first thread's recent updates. The result would be an inconsistent process restart.

To address the need for checkpoint and restart synchronization, we can introduce a barrier operation based on the low-level semaphore mechanism. If  $n$  threads are executing in the process, the barrier operation blocks the first  $n-1$  calling threads, and when the  $n^{\text{th}}$  call on the barrier is performed, all threads resume. At each poll point, if a checkpoint has been requested but is not yet in progress, the threads perform a barrier operation. This has the effect of causing all threads to suspend normal execution before any threads begin producing a stack description. Similarly, at restart time, when each thread reaches its last stack frame (a point in recovery at which the transformed code already performs a **PIL\_DoneRestart**), a barrier operation is performed. This has the effect of ensuring that all threads have fully recovered their stack state before any threads continue normal execution.

We have now described a design for the automatic checkpoint and recovery of multithreaded programs, but have not yet addressed the issue of synchronization constructs. Recall, in our simple threads interface we provided counting semaphores. Thus, in addition to the basic thread control operations that we have already described, semaphore operations must also be available at the wrapper level. Furthermore, the state of all semaphores must be captured and recovered at checkpoint and restart time, respectively.

To perform state capture and recovery in the presence of synchronization operations, the wrapper semaphore functions maintain a list of semaphores in use within the process. Each record on the list records the wrapper-level identification number for the semaphore, the value of the semaphore, and the identifiers of two low-level semaphores that will be used to implement this wrapper semaphore. One of the low level semaphores will be used to synchronize access to the data in the semaphore description record, and one will be used to block calling threads inside **semaphore\_p** operations when necessary. A wrapper-level semaphore description record is



depicted in Figure 8.14. The wrapper semaphore creation routine simply allocates a new record of

---

```

struct PIL_Semaphore {
    int    id;                Unique identifier for this semaphore.
    int    value;            Current value of the semaphore.
    int    mutex;           Low-level semaphore protecting this record's data.
    int    blocked_queue;   Low-level semaphore on which to block P operations.
};

```

---

**Figure 8.14:** Wrapper semaphore record.

this type, initializes the unique identifier and value fields, and uses the low-level `semaphore_create` operation to initialize the data-access protection semaphore (`mutex`) to 1, and the queue semaphore (`blocked_queue`) to 0.

The operation of the wrapper `semaphore_p` operation, depicted in Figure 8.15 has two modes of operation. In the basic mode, the function is called when a restart is not in progress, in which case the value of the semaphore is decremented, and the process is blocked if necessary. In

---

```

void PIL_Semaphore_p(int sem_id) {
    PIL_semaphore *sem;
    sem = lookup_sem(sem_list, sem_id);    Look up the semaphore record.
    if(!PIL_ChkptStatus&PIL_RestoreNow) { If a restart is not in progress, then
        semaphore_p(sem->mutex);           protect the record data,
        sem->value--;                       decrement the semaphore value,
        if(sem->value > 0) {                and check if this call should block.
            semaphore_v(sem->mutex);
            return;
        }
        semaphore_v(sem->mutex);
    }
    semaphore_p(sem->blocked_queue);       Block the calling thread.
    if(PIL_ChkptStatus&PIL_ChkptNow) {    Check for checkpoint request.
        PIL_Barrier();
        PIL_ChkptStatus |= PIL_ChkptInProgress;
    }
}

```

---

**Figure 8.15:** Wrapper semaphore P operation.

the absence of checkpoint/restart activity, this mode of operation in conjunction with the wrapper `semaphore_v` operation depicted in Figure 8.16 provides standard counting semaphore semantics.

---

```

void PIL_Semaphore_v(int sem_id) {
    PIL_semaphore *sem;
    sem = lookup_sem(sem_list, sem_id);    Look up the semaphore record.
    semaphore_p(sem->mutex);
    if(sem->value < 0)                      If threads are blocked,
        semaphore_v(sem->blocked_queue); then wake one up.
    sem->value++;                            Increment the semaphore value.
}

```

---

**Figure 8.16:** Wrapper semaphore V operation.

These semaphore wrapper routines must also support state capture and recovery when threads are blocked on `semaphore_p` calls. For example, suppose a checkpoint request arrives while a thread is blocked, and all other threads reach poll points before calling the `semaphore_v` operations that are necessary to allow the thread to continue normal execution. Without extra support, the result would be deadlock when the other threads attempted to perform a barrier synchronization at the poll points. To address this issue, we must add extra functionality to the wrapper module checkpoint and restart handler to capture and recover the state of all semaphores.

To save the state of a process's semaphores, the checkpoint handler introduced above would iterate over the semaphore list, saving the unique identifier and value of each semaphore. The values of the low-level semaphores need not be saved, as they will be overwritten at restart time. In addition to saving these values, the checkpoint event handler also un-blocks all threads that are blocked in `semaphore_p` operations. This is achieved by calling `semaphore_v` on each semaphore's `blocked_queue` low-level semaphore and incrementing that semaphore's `value` field until it is non-negative. Note in the wrapper `semaphore_p` operation depicted in Figure 8.15, after the call blocks it polls for a checkpoint request. Thus, when the event handler releases the thread, it immediately encounters a poll point to which the thread will be restored. Note also in

Figure 8.15, when the wrapper **semaphore\_p** operation is entered during a state recovery, the thread simply re-blocks itself on the wrapper semaphore's **blocked\_queue**. Thus, with support added to the checkpoint event handler to wake up blocked threads to service a checkpoint request, and support added to the restart handler to recover and re-initialize the active semaphore list, the standard Process Introspection mechanism can be used to recover the state of thread synchronization primitives.

# Chapter 9

## Conclusions and Future Directions

In this dissertation we have presented Process Introspection, a novel design and implementation of a heterogeneous process state capture and recovery mechanism based on the modification of programs to render them self-describing and self-recovering. In this chapter we review the fundamental contributions associated with this work. Research thus far on Process Introspection raises as many directions for further inquiry as it answers existing questions. Thus, in Section 9.2 we examine possible directions for future work related to Process Introspection.

### 9.1 Contributions

First and foremost, in this dissertation we have described a new technique for capturing and restoring the state of a process in a heterogeneous environment. We have described this mechanism on an abstract, general level, and have defined and demonstrated its correctness on that level. Furthermore, we have implemented a complete working prototype of the described mechanism, including a C compiler and run-time support libraries, and we have demonstrated this implementation on representative benchmarks and on real, non-trivial, production applications.

Our experiments using the prototype implementation of the system have produced encouraging results. At the most basic level, our experiences prove that Process Introspection as described in this dissertation can be applied automatically and correctly to significant applications. Furthermore, we find that relatively straightforward automatic poll point placement policies can achieve acceptable levels of incurred overhead while at the same time providing good performance in terms of average checkpoint-request wait time. This result is important—it is the fundamental reason that Process Introspection can be supported by tools that are not unduly complex, yet allow automatic application of this state capture and recovery mechanism without programmer interven-

tion or significant introduced cost.

Beyond good performance, our system provides additional attractive features. The core internal state capture mechanism described is highly portable, requiring no special run-time system support or non-portable code (e.g. assembly language routines). 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—besides being independent of any specific system support, it does not dictate any particular programming style, data structuring techniques, or other artificial limitations on the programs that can be supported.

## 9.2 Future Work

**Integration**—The current Process Introspection implementation is basically a stand-alone package for sequential tasks; state capture and recovery is directed explicitly by an interactive user. A first obvious area for future work is integration into existing distributed systems. We plan to adapt the system for use in the Legion [52] wide-area, object-oriented distributed system, and are also investigating integration into a PVM [34] or MPI [41] system. Adaptability to various system environments is supported explicitly by our PIL API, which provides a medium for APrIL-transformed modules and hand-coded system-interface wrapper modules to interoperate.

**Extensions**—In Chapter 8 we described designs for extensions to the current Process Introspection system to support additional programming languages and multithreaded programming. Implementation of these designs will be the subject of future work to address questions relating to the performance, portability, and complexity of these extensions. For example, we have found that poll point placement using the described heuristic policies generally offers points of acceptable performance/overhead trade-off. Would the same result hold for APrIL-transformed programs compiled by optimizing Fortran compilers, or would different heuristic policies be required? Supporting threads also leads to new questions, such as what are the performance penalties associated

with state capture synchronization among threads? In a situation in which all threads are encountering poll points with approximately equal frequencies, we do not expect that the expense of a barrier at state capture time would introduce significant cost. However, implementation and empirical study could determine if this intuitive argument is correct.

**Performance Enhancements**—In Chapter 7, we identified a number of possible avenues for improving the current APrIL implementation, especially with respect to performance of applications consisting of separately compiled modules. The addition of a “project” abstraction that would allow cross-module application of the APrIL mandatory poll point elimination optimization and would reduce the generation of redundant type and global registration code is the subject of future work. Quantifying the benefits of this enhancement will further delineate the performance obtainable through automatic application of Process Introspection.

**Poll Point Placement Policy Selection**—The performance study in Chapter 7 demonstrated that for a given application, one or more poll point placement policies supported by APrIL can generally provide good performance and low introduced overhead. However, the policies that offer good performance vary among applications, as do users’ desire to trade off performance overhead for state capture request wait-time, or vice-versa. This leads to a policy selection problem—presented with a new application to compile, which poll point policy should a programmer choose? Our performance study has provided strong evidence that selecting from among a very small number of relatively conservative policies is sufficient, but the user is still left with the selection. In future work, we will investigate automating the policy selection process and/or enhancing APrIL to determine appropriate placements dynamically based on higher-level user specifications (e.g. a user might invoke APrIL specifying, “no more than  $n\%$  overhead, no greater than  $x$  ms. between poll points”). A possible approach to this problem might be based on program profiling. Given information about a program’s run-time characteristics (as opposed to just working from source code), a policy would have more information on which to base placement decisions. For example,

currently, placement into loops is based on static characteristics such as the number of statements in the loop body. Run-time statistics such as the actual time spent on average in a loop's body, and the number of times a loop's body was executed could provide a stronger basis for placement selection.

**State Capture Optimizations**—Depending on the intended use of a process state capture and recovery mechanism, various optimizations to the state capture operation are possible. For example, if state capture is used for checkpointing, it is sometimes desirable to support incremental checkpoints. When state capture is performed, only the changed elements of the state are recorded, thus saving space and capture time. In homogeneous systems it is natural to perform this optimization on a page basis, writing only those pages that have become dirty since the last checkpoint. In Process Introspection, a similar optimization can be performed on a state element basis. Memory regions registered with the PIL could be written to the checkpoint only if they had been modified since the last state capture operation. This scheme is straightforward to implement in modern operating systems, which typically support a mechanism for causing a signal to be delivered to a process when certain marked memory regions are accessed. A similar state capture enhancement is sometimes employed for the purposes of low-latency process migration. In these schemes, only the most immediately needed data is transferred to a migrated process so that it can begin execution as quickly as possible after arriving at its target site. The rest of the process state can be transferred later as necessary. A similar technique is possible with Process Introspection. The PIL could be modified to first transfer the minimal data needed to reinstantiate the process's stack and reallocate the process's memory regions. The data associated with top-most stack frames could then be transferred, followed by the data for lower stack frames and other memory regions as needed. The detailed design and implementation of these important state capture optimizations will be the subject of future work.

In addition to the enhancements and extensions we have described, Process Introspection

introduces the possibility of a great deal of additional research in using the system. For example, the use of Process Introspection to develop a truly heterogeneous batch queuing system (e.g. heterogeneous Condor) is one possibility. The development of load balancing policies for heterogeneous PVM networks using Process Introspection is another. Use of Process Introspection as a tool for building higher-level heterogeneous systems will undoubtedly provide insight into added features or improvements in the design. Beyond system development, the application of Process Introspection to additional applications will also provide important feedback into the design refinement process, and will help characterize further the performance of the system.



# Appendix A

## Performance Data

This appendix contains the full results of the performance study discussed in Chapter 7. The tables presented in this appendix fall into the following categories:

- *Execution times*—Presents execution times for a given program, compiled with and without Process Introspection transformations, and with and without compiler optimizations. Each entry contains the mean and standard deviation of 16 runs performed on each of five test platforms. Times are reported in seconds, and were measured using the C `gettimeofday` library routine.
- *Overhead and optimizer effectiveness*—For each poll point placement policy examined,  $p$ , presents the overhead for the non-optimized program,  $O_{norm,p}$ , the optimizer effectiveness  $E$ , and the overhead for the optimized program,  $O_{opt,p}$ , as defined in Section 7.1.1. Speedups due to optimizations,  $S_{ntrans}$  and  $S_{trans,p}$  as defined in Section 7.1.1, are also presented.
- *Poll point counts*—For each poll point placement policy examined, the number of poll points encountered in a complete execution of the program is presented. These were obtained using versions of the applications transformed by APRIL to increment a counter at each poll point.
- *Average poll point interval*—For each poll point placement policy and each test platform, the average interval between poll point encounters computed using Equation 7.6 as defined in Section 7.1.1 is reported. Times are presented in milliseconds.
- *Time to checkpoint/restart*—For each test platform, the time to create a checkpoint and the times to perform restarts from checkpoints created on each of the test platforms are presented. Times are presented in milliseconds, and were measured using the `gettimeofday` C library routine. A special version of the PIL was used to mark the time that the checkpoint or restart was begun and finished. All data disk I/O was performed using a local disk.

## A.1 Basic Numerical Applications

### A.1.1 Matrix Multiply

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
No trans	5.26	0.17	4.56	0.17	18.18	0.09	13.14	0.12	24.77	0.29
No trans, opt.	4.21	0.06	3.51	0.04	9.34	0.04	4.84	0.04	14.05	0.08
0	5.48	0.20	4.81	0.10	18.17	0.08	13.22	0.21	25.06	0.26
0, optimized	4.23	0.14	3.70	0.18	9.40	0.07	4.92	0.07	14.09	0.08
1	5.50	0.12	4.70	0.13	18.19	0.09	13.39	0.10	25.34	0.58
1, optimized	4.29	0.12	3.65	0.12	9.40	0.05	5.05	0.05	14.06	0.10
2	5.46	0.09	4.67	0.15	18.26	0.20	13.69	0.62	25.73	0.95
2, optimized	4.41	0.11	3.67	0.10	9.42	0.03	5.02	0.26	14.00	0.07
3	5.53	0.11	4.66	0.13	18.23	0.18	14.54	0.24	25.27	0.57
3, optimized	4.65	0.26	3.68	0.10	9.43	0.04	5.61	0.10	14.07	0.08
4	5.39	0.13	4.78	0.16	18.22	0.19	13.50	0.37	24.86	0.10
4, optimized	4.57	0.18	3.67	0.10	9.41	0.04	5.51	0.12	14.00	0.06
5	5.84	0.16	4.87	0.08	18.18	0.04	13.42	0.08	24.87	0.16
5, optimized	4.82	0.15	3.64	0.07	9.44	0.04	5.59	0.07	14.05	0.05
11	6.15	0.16	4.88	0.05	18.21	0.20	13.86	0.55	25.29	0.45
1, optimized	4.69	0.16	3.63	0.06	9.44	0.04	5.65	0.25	14.00	0.04
13	6.12	0.12	4.90	0.04	18.21	0.20	14.53	0.10	24.92	0.29
13, optimized	4.56	0.37	3.66	0.08	9.44	0.05	5.11	0.11	13.96	0.06
21	6.76	0.14	4.87	0.06	18.33	0.34	15.93	0.29	26.17	0.11
21, optimized	5.08	0.16	3.72	0.02	9.43	0.05	13.98	0.14	22.10	0.04

Table A.1: Matrix multiply, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
S <sub>ntrans</sub>	1.25		1.30		1.95		2.71		1.76	
O <sub>norm,0</sub>	4.2%	0.5%	5.6%	5.4%	0.0%	0.6%	0.6%	1.6%	1.2%	0.3%
O <sub>opt,0</sub>										
S <sub>trans,0</sub>	1.30	1.04	1.30	1.00	1.93	0.99	2.69	0.99	1.78	1.01
E <sub>0</sub>										
O <sub>norm,1</sub>	4.7%	2.0%	3.2%	4.0%	0.1%	0.6%	1.9%	4.3%	2.3%	0.1%
O <sub>opt,1</sub>										
S <sub>trans,1</sub>	1.28	1.03	1.29	0.99	1.93	0.99	2.65	0.98	1.80	1.02
E <sub>1</sub>										
O <sub>norm,2</sub>	3.9%	4.8%	2.6%	4.7%	0.5%	0.9%	4.2%	3.8%	3.9%	-0.4%
O <sub>opt,2</sub>										
S <sub>trans,2</sub>	1.24	0.99	1.27	0.98	1.94	1.00	2.72	1.00	1.84	1.04
E <sub>2</sub>										
O <sub>norm,3</sub>	5.2%	10.5%	2.3%	4.9%	0.3%	1.0%	10.7%	15.8%	2.0%	0.2%
O <sub>opt,3</sub>										
S <sub>trans,3</sub>	1.19	0.95	1.27	0.98	1.93	0.99	2.59	0.96	1.80	1.02
E <sub>3</sub>										
O <sub>norm,4</sub>	2.5%	8.7%	5.0%	4.6%	0.3%	0.8%	2.7%	13.7%	0.3%	-0.3%
O <sub>opt,4</sub>										
S <sub>trans,4</sub>	1.18	0.94	1.30	1.00	1.94	0.99	2.45	0.90	1.77	1.01
E <sub>4</sub>										
O <sub>norm,5</sub>	11.1%	14.7%	6.9%	3.6%	0.0%	1.0%	2.1%	15.4%	0.4%	0.0%
O <sub>opt,5</sub>										
S <sub>trans,5</sub>	1.21	0.97	1.34	1.03	1.93	0.99	2.40	0.89	1.77	1.00
E <sub>5</sub>										
O <sub>norm,11</sub>	16.9%	11.5%	7.1%	3.4%	0.2%	1.1%	5.5%	16.7%	2.1%	-0.3%
O <sub>opt,11</sub>										
S <sub>trans,11</sub>	1.31	1.05	1.34	1.04	1.93	0.99	2.45	0.90	1.81	1.02
E <sub>11</sub>										
O <sub>norm,13</sub>	16.4%	8.4%	7.5%	4.2%	0.2%	1.1%	10.6%	5.6%	0.6%	-0.7%
O <sub>opt,13</sub>										
S <sub>trans,13</sub>	1.34	1.07	1.34	1.03	1.93	0.99	2.84	1.05	1.79	1.01
E <sub>13</sub>										
O <sub>norm,21</sub>	28.6%	20.8%	6.8%	6.0%	0.8%	1.0%	21.2%	188.8%	5.6%	57.3%
O <sub>opt,21</sub>										
S <sub>trans,21</sub>	1.33	1.06	1.31	1.01	1.94	1.00	1.14	0.42	1.18	0.67
E <sub>21</sub>										

Table A.2: Matrix multiply, overhead and optimizer effectiveness.

Policy	Poll Points
0	1
1	65793
2	65535
3	131329
4	65793
5	131329
11	131329
13	131329
21	16908545

Table A.3: Matrix multiply, poll point counts.

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	4225.1000	3699.0000	9396.6000	4920.4000	14088.2000
1	0.0652	0.0555	0.1429	0.0767	0.2137
2	0.0673	0.0561	0.1438	0.0766	0.2136
3	0.0354	0.0280	0.0718	0.0427	0.1072
4	0.0695	0.0558	0.1431	0.0837	0.2129
5	0.0367	0.0277	0.0719	0.0425	0.1070
11	0.0357	0.0276	0.0719	0.0430	0.1066
13	0.0347	0.0279	0.0719	0.0389	0.1063
21	0.0003	0.0002	0.0006	0.0008	0.0013

Table A.4: Matrix multiply, average poll point interval (milliseconds).

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chckpt size</i>	1573018		1573018		1573018		1573018		1573018	
$t_{chckpt}$	32.59	0.84	47.31	1.87	324.55	6.80	95.40	1.12	253.90	8.89
$t_{restart.sparc}$	104.05	1.77	131.65	0.47	176.77	0.96	110.31	1.04	259.51	9.30
$t_{restart.mips}$	101.69	2.04	130.62	2.57	176.75	1.49	111.34	2.51	261.18	12.07
$t_{restart.x86}$	54.24	1.11	99.84	3.59	258.50	2.82	177.79	6.53	346.44	2.99
$t_{restart.rs/6000}$	101.93	1.61	131.70	1.13	177.28	1.82	112.66	2.45	258.87	2.80
$t_{restart.alpha}$	55.30	1.19	97.82	2.19	268.43	24.69	179.64	6.38	346.87	2.11

Table A.5: Matrix multiply, time to checkpoint/restart (milliseconds).

## A.1.2 Gauss-Seidel

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
No trans	12.37	0.64	10.35	0.05	44.78	0.19	40.97	1.06	65.16	0.54
No trans, opt.	4.83	0.07	5.00	0.04	21.43	0.21	26.00	0.60	23.80	0.26
0	12.60	0.19	10.40	0.03	45.40	0.26	41.01	1.14	65.11	0.74
0, optimized	4.84	0.05	5.02	0.02	22.97	0.16	25.72	0.22	23.92	0.27
1	12.88	0.44	10.61	0.68	45.51	0.32	41.69	1.47	65.40	1.04
1, optimized	4.92	0.04	5.03	0.05	23.20	0.17	26.44	0.45	23.79	0.38
2	12.72	0.16	10.40	0.05	45.54	0.26	42.35	1.37	65.18	0.48
2, optimized	4.91	0.04	5.04	0.05	23.23	0.22	26.00	0.66	25.15	1.37
3	12.88	0.20	10.41	0.07	45.52	0.26	43.22	2.01	65.17	0.53
3, optimized	4.94	0.03	5.01	0.06	23.22	0.23	26.48	1.89	24.65	0.81
4	13.29	0.19	10.90	0.07	48.86	0.29	51.57	1.70	68.46	0.47
4, optimized	5.26	0.04	5.77	0.04	24.02	0.20	30.95	0.87	28.53	0.25
5	13.35	0.73	10.86	0.03	48.99	0.24	50.73	1.77	68.71	1.28
5, optimized	5.33	0.04	5.78	0.05	24.00	0.17	32.00	1.19	28.55	0.19
11	12.85	0.40	10.41	0.07	45.56	0.33	42.04	1.68	65.35	0.90
1, optimized	5.17	0.31	5.01	0.03	23.19	0.20	26.11	1.04	23.50	0.43
13	13.33	0.72	10.77	0.05	49.00	0.25	51.19	1.77	69.03	0.95
13, optimized	5.30	0.09	5.74	0.10	24.18	0.56	32.21	1.36	28.38	0.08
21	13.56	0.18	10.87	0.05	52.26	0.18	55.76	1.93	73.02	1.56
21, optimized	5.99	0.21	5.92	0.11	29.20	0.19	44.78	1.69	44.82	0.33

Table A.6: Gauss-Seidel, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
S <sub>ntrans</sub>	2.56		2.07		2.09		1.58		2.74	
O <sub>norm,0</sub> O <sub>opt,0</sub>	1.8%	0.2%	0.5%	0.4%	1.4%	7.2%	0.1%	-1.1%	-0.1%	0.5%
S <sub>trans,0</sub> E <sub>0</sub>	2.60	1.02	2.07	1.00	1.98	0.95	1.59	1.01	2.72	0.99
O <sub>norm,1</sub> O <sub>opt,1</sub>	4.0%	1.7%	2.5%	0.6%	1.6%	8.2%	1.8%	1.7%	0.4%	0.0%
S <sub>trans,1</sub> E <sub>1</sub>	2.62	1.02	2.11	1.02	1.96	0.94	1.58	1.00	2.75	1.00
O <sub>norm,2</sub> O <sub>opt,2</sub>	2.8%	1.7%	0.5%	0.8%	1.7%	8.4%	3.4%	0.0%	0.0%	5.7%
S <sub>trans,2</sub> E <sub>2</sub>	2.59	1.01	2.06	1.00	1.96	0.94	1.63	1.03	2.59	0.95
O <sub>norm,3</sub> O <sub>opt,3</sub>	4.1%	2.2%	0.6%	0.2%	1.7%	8.4%	5.5%	1.8%	0.0%	3.6%
S <sub>trans,3</sub> E <sub>3</sub>	2.61	1.02	2.08	1.00	1.96	0.94	1.63	1.04	2.64	0.97
O <sub>norm,4</sub> O <sub>opt,4</sub>	7.4%	8.7%	5.4%	15.4%	9.1%	12.1%	25.9%	19.0%	5.1%	19.9%
S <sub>trans,4</sub> E <sub>4</sub>	2.53	0.99	1.89	0.91	2.03	0.97	1.67	1.06	2.40	0.88
O <sub>norm,5</sub> O <sub>opt,5</sub>	7.8%	10.3%	5.0%	15.7%	9.4%	12.0%	23.8%	23.1%	5.5%	20.0%
S <sub>trans,5</sub> E <sub>5</sub>	2.50	0.98	1.88	0.91	2.04	0.98	1.59	1.01	2.41	0.88
O <sub>norm,11</sub> O <sub>opt,11</sub>	3.8%	6.9%	0.6%	0.3%	1.8%	8.2%	2.6%	0.4%	0.3%	-1.3%
S <sub>trans,11</sub> E <sub>11</sub>	2.49	0.97	2.08	1.00	1.96	0.94	1.61	1.02	2.78	1.02
O <sub>norm,13</sub> O <sub>opt,13</sub>	7.7%	9.6%	4.1%	14.8%	9.4%	12.9%	24.9%	23.9%	5.9%	19.3%
S <sub>trans,13</sub> E <sub>13</sub>	2.52	0.98	1.88	0.91	2.03	0.97	1.59	1.01	2.43	0.89
O <sub>norm,21</sub> O <sub>opt,21</sub>	9.6%	24.0%	5.0%	18.4%	16.7%	36.3%	36.1%	72.2%	12.1%	88.3%
S <sub>trans,21</sub> E <sub>21</sub>	2.26	0.88	1.84	0.89	1.79	0.86	1.25	0.79	1.63	0.60

Table A.7: Gauss-Seidel, overhead and optimizer effectiveness.

Policy	Poll Points
0	11685
1	461596
2	473838
3	473838
4	38786389
5	38786389
11	473838
13	38786389
21	74335201

Table A.8: Gauss-Seidel, poll point counts.

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	0.41444	0.42966	1.96561	2.20153	2.04732
1	0.01066	0.01090	0.05025	0.05729	0.05153
2	0.01037	0.01063	0.04902	0.05487	0.05307
3	0.01043	0.01057	0.04900	0.05588	0.05203
4	0.00014	0.00015	0.00062	0.00080	0.00074
5	0.00014	0.00015	0.00062	0.00083	0.00074
11	0.01091	0.01058	0.04894	0.05510	0.04959
13	0.00014	0.00015	0.00062	0.00083	0.00073
21	0.00008	0.00008	0.00039	0.00060	0.00060

Table A.9: Gauss-Seidel, average poll point interval (milliseconds).

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chckpt size</i>	102613		102613		102613		102613		102613	
<i>t<sub>chkpt</sub></i>	2.56	0.47	4.05	0.37	27.57	0.70	10.67	0.46	13.25	0.45
<i>t<sub>restart.sparc</sub></i>	6.35	0.49	7.57	0.18	14.21	0.14	12.14	0.25	18.00	0.33
<i>t<sub>restart.mips</sub></i>	5.98	0.32	7.41	0.18	14.14	0.07	12.38	0.62	18.04	0.31
<i>t<sub>restart.x86</sub></i>	3.54	0.47	6.13	0.10	20.05	0.61	16.55	0.79	23.22	0.27
<i>t<sub>restart.rs/6000</sub></i>	6.10	0.42	7.51	0.15	14.16	0.11	12.17	0.22	18.37	0.63
<i>t<sub>restart.alpha</sub></i>	3.54	0.47	6.11	0.09	19.67	0.16	16.20	0.84	25.21	0.32

Table A.10: Gauss-Seidel, time to checkpoint/restart (milliseconds).

## A.1.3 Quicksort

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
No trans	7.42	0.02	8.34	0.02	22.61	0.17	24.86	0.97	32.30	0.31
No trans, opt.	2.26	0.01	4.84	0.09	9.82	0.12	12.42	0.42	14.49	0.15
0	7.48	0.03	8.38	0.05	24.38	0.17	25.56	0.72	33.58	0.31
0, optimized	2.39	0.04	4.94	0.05	11.53	0.04	13.93	0.81	15.18	0.12
1	7.72	0.06	8.61	0.04	26.27	0.16	28.55	0.95	35.12	0.42
1, optimized	2.66	0.04	5.29	0.03	14.48	0.19	16.99	0.47	15.88	0.18
2	7.71	0.04	8.62	0.05	26.34	0.34	27.97	0.87	35.19	0.43
2, optimized	2.67	0.04	5.28	0.03	14.41	0.10	17.96	0.18	15.96	0.10
3	7.70	0.04	8.63	0.06	26.28	0.22	28.58	0.94	35.40	0.43
3, optimized	2.66	0.03	5.30	0.03	14.44	0.19	16.93	0.38	15.98	0.02
4	7.72	0.05	8.71	0.07	26.46	0.17	28.27	0.87	35.23	0.38
4, optimized	2.68	0.02	5.31	0.03	14.61	0.18	17.07	0.11	16.01	0.17
5	7.70	0.02	8.68	0.06	26.47	0.19	28.81	0.98	35.71	0.41
5, optimized	2.69	0.04	5.32	0.04	14.56	0.05	18.30	0.59	16.15	0.26
11	8.19	0.05	8.60	0.03	26.21	0.04	28.07	0.86	36.30	0.25
1, optimized	2.73	0.26	5.30	0.02	14.44	0.17	17.07	0.51	16.42	0.21
13	8.18	0.04	8.67	0.03	26.50	0.25	28.72	0.98	36.17	0.67
13, optimized	2.77	0.30	5.29	0.02	14.62	0.18	17.04	0.11	16.01	0.29
21	8.38	0.03	8.96	0.05	36.06	0.16	34.32	1.78	41.17	1.70
21, optimized	3.13	0.05	6.03	0.02	17.14	0.06	26.50	0.63	19.53	0.16

Table A.11: Quicksort, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
S <sub>ntrans</sub>	3.28		1.72		2.30		2.00		2.23	
O <sub>norm,0</sub> O <sub>opt,0</sub>	0.8%	5.6%	0.5%	2.0%	7.8%	17.4%	2.8%	12.1%	4.0%	4.8%
S <sub>trans,0</sub> E <sub>0</sub>	3.13	0.95	1.70	0.99	2.11	0.92	1.84	0.92	2.21	0.99
O <sub>norm,1</sub> O <sub>opt,1</sub>	4.1%	17.6%	3.3%	9.2%	16.2%	47.5%	14.9%	36.7%	8.7%	9.6%
S <sub>trans,1</sub> E <sub>1</sub>	2.90	0.88	1.63	0.95	1.81	0.79	1.68	0.84	2.21	0.99
O <sub>norm,2</sub> O <sub>opt,2</sub>	3.9%	17.9%	3.3%	9.1%	16.5%	46.8%	12.5%	44.5%	8.9%	10.2%
S <sub>trans,2</sub> E <sub>2</sub>	2.89	0.88	1.63	0.95	1.83	0.79	1.56	0.78	2.21	0.99
O <sub>norm,3</sub> O <sub>opt,3</sub>	3.8%	17.8%	3.5%	9.5%	16.2%	47.2%	15.0%	36.3%	9.6%	10.3%
S <sub>trans,3</sub> E <sub>3</sub>	2.89	0.88	1.63	0.95	1.82	0.79	1.69	0.84	2.22	0.99
O <sub>norm,4</sub> O <sub>opt,4</sub>	4.0%	18.6%	4.4%	9.7%	17.0%	48.9%	13.7%	37.4%	9.1%	10.6%
S <sub>trans,4</sub> E <sub>4</sub>	2.88	0.88	1.64	0.95	1.81	0.79	1.66	0.83	2.20	0.99
O <sub>norm,5</sub> O <sub>opt,5</sub>	3.8%	18.9%	4.1%	9.9%	17.1%	48.3%	15.9%	47.3%	10.5%	11.5%
S <sub>trans,5</sub> E <sub>5</sub>	2.86	0.87	1.63	0.95	1.82	0.79	1.57	0.79	2.21	0.99
O <sub>norm,11</sub> O <sub>opt,11</sub>	10.4%	20.7%	3.1%	9.3%	15.9%	47.1%	12.9%	37.4%	12.4%	13.3%
S <sub>trans,11</sub> E <sub>11</sub>	3.00	0.91	1.62	0.94	1.82	0.79	1.64	0.82	2.21	0.99
O <sub>norm,13</sub> O <sub>opt,13</sub>	10.3%	22.6%	4.0%	9.2%	17.2%	48.9%	15.6%	37.2%	12.0%	10.5%
S <sub>trans,13</sub> E <sub>13</sub>	2.95	0.90	1.64	0.95	1.81	0.79	1.69	0.84	2.26	1.01
O <sub>norm,21</sub> O <sub>opt,21</sub>	13.0%	38.3%	7.4%	24.5%	59.5%	74.7%	38.1%	113.3%	27.5%	34.9%
S <sub>trans,21</sub> E <sub>21</sub>	2.68	0.82	1.48	0.86	2.10	0.91	1.30	0.65	2.11	0.95

Table A.12: Quicksort, overhead and optimizer effectiveness.

Policy	Poll Points
0	18874363
1	32129457
2	32129457
3	32129457
4	34226609
5	34226609
11	32129457
13	34226609
21	95791100

Table A.13: Quicksort, poll point counts.

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	0.000127	0.000262	0.000611	0.000738	0.000804
1	0.000083	0.000165	0.000451	0.000529	0.000494
2	0.000083	0.000164	0.000449	0.000559	0.000497
3	0.000083	0.000165	0.000450	0.000527	0.000497
4	0.000078	0.000155	0.000427	0.000499	0.000468
5	0.000079	0.000155	0.000425	0.000535	0.000472
11	0.000085	0.000165	0.000450	0.000531	0.000511
13	0.000081	0.000155	0.000427	0.000498	0.000468
21	0.000033	0.000063	0.000179	0.000277	0.000204

Table A.14: Quicksort, average poll point interval (milliseconds).

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chckpt size</i>	8388716		8388716		8388716		8388716		8388716	
$t_{chckpt}$	161.19	2.32	303.72	2.11	2064	25.99	614.66	4.50	1771.56	15.35
$t_{restart.sparc}$	587.41	11.64	748.84	3.80	1041	86.63	602.10	4.35	1308.46	6.41
$t_{restart.mips}$	589.36	10.83	751.45	10.66	1116	115.68	608.42	6.75	1420.06	57.84
$t_{restart.x86}$	305.57	4.02	570.50	15.10	1516	76.38	1039.50	15.33	2125.57	13.13
$t_{restart.rs/6000}$	583.08	4.46	748.07	12.71	1011	6.14	605.35	5.89	1473.02	17.30
$t_{restart.alpha}$	309.00	10.09	560.69	9.74	1457	23.49	1041.41	9.04	2047.77	116.80

Table A.15: Quicksort, time to checkpoint/restart (milliseconds).

### A.1.4 Gaussian Elimination

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
No trans	12.30	0.05	14.69	0.01	30.48	0.18	31.38	1.12	51.76	0.51
No trans, opt.	5.12	0.04	8.23	0.03	8.99	0.04	28.79	1.06	17.91	0.26
0	12.39	0.04	16.43	0.05	30.55	0.19	33.29	0.95	52.29	0.76
0, optimized	5.17	0.04	8.23	0.05	9.14	0.18	30.04	1.07	17.68	0.65
1	12.36	0.05	16.42	0.04	30.60	0.19	33.79	0.59	52.29	0.62
1, optimized	5.13	0.04	8.26	0.03	9.09	0.05	29.02	1.06	18.41	0.88
2	12.32	0.04	16.47	0.02	30.58	0.17	32.92	0.59	51.96	0.40
2, optimized	5.14	0.05	8.27	0.04	9.13	0.17	29.76	1.06	17.90	0.78
3	12.37	0.05	16.43	0.05	30.65	0.23	33.61	1.09	51.98	0.36
3, optimized	5.14	0.06	8.28	0.05	9.11	0.03	29.34	1.04	20.25	0.81
4	12.36	0.03	16.49	0.04	30.63	0.20	31.95	0.23	52.23	0.61
4, optimized	5.19	0.05	8.27	0.03	9.08	0.06	29.47	0.95	18.91	1.56
5	12.38	0.07	16.44	0.04	30.61	0.25	31.99	0.30	52.69	1.03
5, optimized	5.19	0.04	8.27	0.04	9.17	0.18	30.05	0.85	21.21	1.05
11	12.32	0.05	16.44	0.03	30.59	0.18	33.38	1.22	53.54	1.37
1, optimized	5.21	0.04	8.26	0.02	9.14	0.06	28.33	0.61	18.56	2.00
13	12.37	0.16	16.44	0.05	30.63	0.29	32.03	0.56	54.08	1.52
13, optimized	5.21	0.04	8.27	0.04	9.17	0.18	29.80	1.00	18.49	1.92
21	13.08	0.06	16.82	0.02	34.33	0.17	39.70	1.30	57.16	1.60
21, optimized	5.79	0.03	8.37	0.03	16.98	0.06	36.33	0.77	30.77	1.85

Table A.16: Gaussian elimination, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
S <sub>ntrans</sub>	2.40		1.79		3.39		1.09		2.89	
O <sub>norm,0</sub> O <sub>opt,0</sub>	0.7%	1.0%	11.8%	0.1%	0.2%	1.7%	6.1%	4.3%	1.0%	-1.3%
S <sub>trans,0</sub> E <sub>0</sub>	2.40	1.00	2.00	1.12	3.34	0.99	1.11	1.02	2.96	1.02
O <sub>norm,1</sub> O <sub>opt,1</sub>	0.5%	0.2%	11.8%	0.4%	0.4%	1.0%	7.7%	0.8%	1.0%	2.8%
S <sub>trans,1</sub> E <sub>1</sub>	2.41	1.00	1.99	1.11	3.37	0.99	1.16	1.07	2.84	0.98
O <sub>norm,2</sub> O <sub>opt,2</sub>	0.1%	0.4%	12.1%	0.5%	0.3%	1.5%	4.9%	3.4%	0.4%	0.0%
S <sub>trans,2</sub> E <sub>2</sub>	2.40	1.00	1.99	1.12	3.35	0.99	1.11	1.01	2.90	1.00
O <sub>norm,3</sub> O <sub>opt,3</sub>	0.5%	0.5%	11.8%	0.7%	0.6%	1.3%	7.1%	1.9%	0.4%	13.0%
S <sub>trans,3</sub> E <sub>3</sub>	2.41	1.00	1.98	1.11	3.37	0.99	1.15	1.05	2.57	0.89
O <sub>norm,4</sub> O <sub>opt,4</sub>	0.4%	1.4%	12.2%	0.5%	0.5%	1.0%	1.8%	2.4%	0.9%	5.6%
S <sub>trans,4</sub> E <sub>4</sub>	2.38	0.99	1.99	1.12	3.37	1.00	1.08	0.99	2.76	0.96
O <sub>norm,5</sub> O <sub>opt,5</sub>	0.6%	1.4%	11.9%	0.5%	0.4%	1.9%	1.9%	4.4%	1.8%	18.4%
S <sub>trans,5</sub> E <sub>5</sub>	2.39	0.99	1.99	1.11	3.34	0.99	1.06	0.98	2.48	0.86
O <sub>norm,11</sub> O <sub>opt,11</sub>	0.1%	1.8%	11.9%	0.4%	0.4%	1.6%	6.4%	-1.6%	3.4%	3.6%
S <sub>trans,11</sub> E <sub>11</sub>	2.36	0.98	1.99	1.11	3.35	0.99	1.18	1.08	2.89	1.00
O <sub>norm,13</sub> O <sub>opt,13</sub>	0.5%	1.8%	11.9%	0.5%	0.5%	2.0%	2.1%	3.5%	4.5%	3.3%
S <sub>trans,13</sub> E <sub>13</sub>	2.37	0.99	1.99	1.11	3.34	0.99	1.07	0.99	2.92	1.01
O <sub>norm,21</sub> O <sub>opt,21</sub>	6.3%	13.1%	14.5%	1.7%	12.6%	88.8%	26.5%	26.2%	10.4%	71.8%
S <sub>trans,21</sub> E <sub>21</sub>	2.26	0.94	2.01	1.13	2.02	0.60	1.09	1.00	1.86	0.64

Table A.17: Gaussian elimination, overhead and optimizer effectiveness.



Policy	Poll Points
0	7
1	132868
2	2052
3	132868
4	3076
5	133892
11	132868
13	133892
21	45528068

Table A.18: Gaussian elimination, poll point counts.

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	738.1286	1176.1143	1305.9000	4291.3429	2525.0429
1	0.0386	0.0622	0.0684	0.2184	0.1385
2	2.5049	4.0288	4.4477	14.5026	8.7243
3	0.0387	0.0623	0.0685	0.2208	0.1524
4	1.6866	2.6888	2.9519	9.5801	6.1485
5	0.0387	0.0617	0.0685	0.2244	0.1584
11	0.0392	0.0622	0.0688	0.2133	0.1397
13	0.0389	0.0617	0.0685	0.2226	0.1381
21	0.0001	0.0002	0.0004	0.0008	0.0007

Table A.19: Gaussian elimination, average poll point interval (milliseconds).

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chckpt size</i>	102613		102613		102613		102613		102613	
$t_{chckpt}$	2.56	0.47	4.05	0.37	27.57	0.70	10.67	0.46	13.25	0.45
$t_{restart.sparc}$	6.35	0.49	7.57	0.18	14.21	0.14	12.14	0.25	18.00	0.33
$t_{restart.mips}$	5.98	0.32	7.41	0.18	14.14	0.07	12.38	0.62	18.04	0.31
$t_{restart.x86}$	3.54	0.47	6.13	0.10	20.05	0.61	16.55	0.79	23.22	0.27
$t_{restart.rs/6000}$	6.10	0.42	7.51	0.15	14.16	0.11	12.17	0.22	18.37	0.63
$t_{restart.alpha}$	3.54	0.47	6.11	0.09	19.67	0.16	16.20	0.84	25.21	0.32

Table A.20: Gaussian elimination, time to checkpoint/restart (milliseconds).

### A.1.5 Conjugate Gradient

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
No trans	14.62	0.06	15.04	0.07	35.64	0.19	28.57	0.32	40.13	0.54
No trans, opt.	6.82	0.22	11.41	0.04	17.32	0.16	22.28	0.25	19.68	0.16
0	14.97	0.22	15.44	0.04	38.57	0.23	30.81	1.25	40.16	0.51
0, optimized	6.90	0.07	11.48	0.07	20.20	0.13	22.82	0.31	19.63	0.21
1	15.02	0.31	15.47	0.06	38.70	0.37	28.75	0.12	40.25	0.47
1, optimized	7.46	0.12	11.48	0.07	20.19	0.03	22.30	0.14	19.62	0.16
2	14.79	0.22	15.49	0.03	38.59	0.24	31.22	1.31	40.61	0.83
2, optimized	7.24	0.23	11.45	0.07	20.26	0.23	23.46	0.70	19.73	0.30
3	14.83	0.35	15.49	0.06	38.60	0.22	29.13	0.89	40.15	0.48
3, optimized	6.96	0.14	11.43	0.06	20.25	0.21	22.65	0.75	19.82	0.16
4	15.86	0.28	16.02	0.04	43.92	0.18	42.73	2.08	44.23	0.44
4, optimized	7.52	0.20	12.69	0.08	21.19	0.19	32.66	0.55	25.05	0.23
5	15.61	0.27	16.07	0.08	43.95	0.26	43.04	2.06	44.85	0.62
5, optimized	7.54	0.21	12.64	0.08	21.20	0.19	32.37	0.26	24.69	0.37
11	14.87	0.27	15.49	0.05	38.59	0.27	29.52	1.19	40.33	0.45
1, optimized	6.84	0.09	11.39	0.10	20.29	0.19	23.16	0.54	19.71	0.23
13	15.60	0.15	16.00	0.07	43.99	0.26	42.48	1.99	45.11	0.79
13, optimized	7.64	0.32	12.68	0.06	21.22	0.17	32.36	0.36	25.10	0.21
21	15.82	0.19	16.01	0.09	45.28	0.25	44.52	2.10	45.61	0.67
21, optimized	7.30	0.25	12.76	0.10	21.30	0.18	34.81	0.67	25.98	0.43

Table A.21: Conjugate gradient, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
S <sub>ntrans</sub>	2.14		1.32		2.06		1.28		2.04	
O <sub>norm,0</sub> O <sub>opt,0</sub>	2.4%	1.2%	2.7%	0.6%	8.2%	16.6%	7.8%	2.4%	0.1%	-0.3%
S <sub>trans,0</sub> E <sub>0</sub>	2.17	1.01	1.34	1.02	1.91	0.93	1.35	1.05	2.05	1.00
O <sub>norm,1</sub> O <sub>opt,1</sub>	2.7%	9.4%	2.9%	0.6%	8.6%	16.5%	0.6%	0.1%	0.3%	-0.3%
S <sub>trans,1</sub> E <sub>1</sub>	2.01	0.94	1.35	1.02	1.92	0.93	1.29	1.01	2.05	1.01
O <sub>norm,2</sub> O <sub>opt,2</sub>	1.2%	6.2%	3.0%	0.3%	8.3%	17.0%	9.3%	5.3%	1.2%	0.3%
S <sub>trans,2</sub> E <sub>2</sub>	2.04	0.95	1.35	1.03	1.90	0.93	1.33	1.04	2.06	1.01
O <sub>norm,3</sub> O <sub>opt,3</sub>	1.5%	2.2%	3.0%	0.2%	8.3%	16.9%	2.0%	1.7%	0.1%	0.7%
S <sub>trans,3</sub> E <sub>3</sub>	2.13	0.99	1.35	1.03	1.91	0.93	1.29	1.00	2.03	0.99
O <sub>norm,4</sub> O <sub>opt,4</sub>	8.5%	10.4%	6.5%	11.2%	23.2%	22.3%	49.6%	46.6%	10.2%	27.3%
S <sub>trans,4</sub> E <sub>4</sub>	2.11	0.98	1.26	0.96	2.07	1.01	1.31	1.02	1.77	0.87
O <sub>norm,5</sub> O <sub>opt,5</sub>	6.8%	10.6%	6.9%	10.8%	23.3%	22.4%	50.6%	45.3%	11.8%	25.4%
S <sub>trans,5</sub> E <sub>5</sub>	2.07	0.97	1.27	0.97	2.07	1.01	1.33	1.04	1.82	0.89
O <sub>norm,11</sub> O <sub>opt,11</sub>	1.8%	0.3%	3.0%	-0.2%	8.3%	17.1%	3.3%	4.0%	0.5%	0.1%
S <sub>trans,11</sub> E <sub>11</sub>	2.18	1.01	1.36	1.03	1.90	0.92	1.27	0.99	2.05	1.00
O <sub>norm,13</sub> O <sub>opt,13</sub>	6.7%	12.1%	6.4%	11.1%	23.4%	22.5%	48.7%	45.3%	12.4%	27.5%
S <sub>trans,13</sub> E <sub>13</sub>	2.04	0.95	1.26	0.96	2.07	1.01	1.31	1.02	1.80	0.88
O <sub>norm,21</sub> O <sub>opt,21</sub>	8.2%	7.0%	6.5%	11.8%	27.0%	23.0%	55.8%	56.2%	13.6%	32.0%
S <sub>trans,21</sub> E <sub>21</sub>	2.17	1.01	1.25	0.95	2.13	1.03	1.28	1.00	1.76	0.86

Table A.22: Conjugate gradient, overhead and optimizer effectiveness.

Policy	Poll Points
0	2223
1	54530
2	54802
3	54802
4	56204979
5	56204979
11	54802
13	56204979
21	66946875

Table A.23: Conjugate gradient, poll point counts.

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	3.10387	5.16433	9.08511	10.26415	8.83198
1	0.13681	0.21060	0.37019	0.40895	0.35980
2	0.13212	0.20898	0.36977	0.42802	0.36011
3	0.12709	0.20863	0.36947	0.41339	0.36164
4	0.00013	0.00023	0.00038	0.00058	0.00045
5	0.00013	0.00022	0.00038	0.00058	0.00044
11	0.12474	0.20783	0.37018	0.42261	0.35963
13	0.00014	0.00023	0.00038	0.00058	0.00045
21	0.00011	0.00019	0.00032	0.00052	0.00039

Table A.24: Conjugate gradient, average poll point interval (milliseconds).

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chckpt size</i>	1280353		1280353		1280353		1280353		1280353	
$t_{chckpt}$	26.98	0.47	46.57	1.84	275.97	5.74	87.82	3.47	222.12	5.29
$t_{restart.sparc}$	82.52	1.38	115.48	4.22	162.07	1.75	101.80	1.07	216.95	11.96
$t_{restart.mips}$	83.01	1.54	113.15	1.01	161.02	1.69	101.58	2.66	228.44	3.69
$t_{restart.x86}$	43.34	1.09	84.60	1.79	227.60	1.19	153.00	4.46	286.75	11.67
$t_{restart.rs/6000}$	82.40	1.46	117.61	7.63	161.04	0.98	100.88	0.52	228.95	3.68
$t_{restart.alpha}$	44.80	1.98	84.60	1.15	227.91	2.45	150.34	1.18	298.82	10.49

Table A.25: Conjugate gradient, time to checkpoint/restart (milliseconds).

## A.2 NAS Benchmarks

### A.2.1 NAS IS Kernel

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
No trans	17.56	0.04	26.79	0.04	68.17	0.29	112.59	0.99	85.40	2.24
No trans, opt.	7.36	0.03	23.05	0.01	49.04	0.26	105.07	2.97	54.01	0.95
0	17.57	0.04	35.94	0.03	69.18	0.36	113.06	3.30	86.68	2.03
0, optimized	7.43	0.04	23.27	0.01	50.42	0.28	107.71	3.45	54.14	1.32
1	17.68	0.29	35.95	0.01	69.17	0.28	113.22	3.79	85.31	1.71
1, optimized	7.47	0.05	23.27	0.01	50.30	0.27	107.70	3.27	54.36	0.92
2	18.03	0.03	35.10	0.03	70.55	0.27	116.98	4.48	87.93	2.02
2, optimized	7.98	0.07	24.16	0.01	52.67	0.22	109.99	3.35	59.41	1.35
3	18.04	0.07	35.11	0.06	70.47	0.28	116.33	3.34	88.79	1.81
3, optimized	7.99	0.04	24.17	0.01	52.68	0.27	109.96	3.45	59.18	1.22
4	19.40	0.07	35.28	0.02	78.34	0.36	122.38	1.86	93.52	1.32
4, optimized	9.25	0.03	24.89	0.02	64.68	0.28	123.97	5.02	59.23	3.04
5	19.38	0.07	35.28	0.02	78.36	0.42	122.78	2.43	93.78	1.94
5, optimized	9.24	0.05	24.90	0.04	64.97	0.96	123.32	3.57	58.49	3.02
11	18.08	0.11	35.10	0.02	70.52	0.30	116.31	3.45	88.34	2.73
1, optimized	8.00	0.05	24.33	0.01	52.70	0.26	109.77	3.30	67.65	0.46
13	19.44	0.06	35.28	0.02	78.32	0.33	123.46	3.83	91.68	2.07
13, optimized	9.27	0.04	25.05	0.02	64.71	0.28	123.36	3.66	59.65	3.27
21	19.44	0.07	35.26	0.02	78.24	0.26	123.95	3.60	90.80	1.95
21, optimized	9.28	0.06	25.06	0.01	64.71	0.30	123.22	3.71	58.21	3.16

Table A.26: NAS IS, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
S <sub>ntrans</sub>	2.39		1.16		1.39		1.07		1.58	
O <sub>norm,0</sub> O <sub>opt,0</sub>	0.1%	1.0%	34.2%	1.0%	1.5%	2.8%	0.4%	2.5%	1.5%	0.2%
S <sub>trans,0</sub> E <sub>0</sub>	2.37	0.99	1.54	1.33	1.37	0.99	1.05	0.98	1.60	1.01
O <sub>norm,1</sub> O <sub>opt,1</sub>	0.7%	1.5%	34.2%	1.0%	1.5%	2.6%	0.6%	2.5%	-0.1%	0.6%
S <sub>trans,1</sub> E <sub>1</sub>	2.37	0.99	1.54	1.33	1.38	0.99	1.05	0.98	1.57	0.99
O <sub>norm,2</sub> O <sub>opt,2</sub>	2.7%	8.6%	31.0%	4.8%	3.5%	7.4%	3.9%	4.7%	3.0%	10.0%
S <sub>trans,2</sub> E <sub>2</sub>	2.26	0.95	1.45	1.25	1.34	0.96	1.06	0.99	1.48	0.94
O <sub>norm,3</sub> O <sub>opt,3</sub>	2.8%	8.6%	31.1%	4.8%	3.4%	7.4%	3.3%	4.7%	4.0%	9.6%
S <sub>trans,3</sub> E <sub>3</sub>	2.26	0.95	1.45	1.25	1.34	0.96	1.06	0.99	1.50	0.95
O <sub>norm,4</sub> O <sub>opt,4</sub>	10.5%	25.7%	31.7%	8.0%	14.9%	31.9%	8.7%	18.0%	9.5%	9.7%
S <sub>trans,4</sub> E <sub>4</sub>	2.10	0.88	1.42	1.22	1.21	0.87	0.99	0.92	1.58	1.00
O <sub>norm,5</sub> O <sub>opt,5</sub>	10.4%	25.6%	31.7%	8.0%	14.9%	32.5%	9.1%	17.4%	9.8%	8.3%
S <sub>trans,5</sub> E <sub>5</sub>	2.10	0.88	1.42	1.22	1.21	0.87	1.00	0.93	1.60	1.01
O <sub>norm,11</sub> O <sub>opt,11</sub>	3.0%	8.8%	31.0%	5.5%	3.4%	7.5%	3.3%	4.5%	3.4%	25.3%
S <sub>trans,11</sub> E <sub>11</sub>	2.26	0.95	1.44	1.24	1.34	0.96	1.06	0.99	1.31	0.83
O <sub>norm,13</sub> O <sub>opt,13</sub>	10.7%	26.0%	31.7%	8.7%	14.9%	32.0%	9.7%	17.4%	7.4%	10.4%
S <sub>trans,13</sub> E <sub>13</sub>	2.10	0.88	1.41	1.21	1.21	0.87	1.00	0.93	1.54	0.97
O <sub>norm,21</sub> O <sub>opt,21</sub>	10.7%	26.1%	31.6%	8.7%	14.8%	32.0%	10.1%	17.3%	6.3%	7.8%
S <sub>trans,21</sub> E <sub>21</sub>	2.10	0.88	1.41	1.21	1.21	0.87	1.01	0.94	1.56	0.99

Table A.27: NAS IS, overhead and optimizer effectiveness.

Policy	Poll Points
0	8388628
1	8388628
2	10485859
3	10485859
4	73441369
5	73441369
11	10485859
13	73441369
21	73441369

Table A.28: NAS IS, poll point counts.

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	0.00089	0.00277	0.00601	0.01284	0.00645
1	0.00089	0.00277	0.00600	0.01284	0.00648
2	0.00076	0.00230	0.00502	0.01049	0.00567
3	0.00076	0.00230	0.00502	0.01049	0.00564
4	0.00013	0.00034	0.00088	0.00169	0.00081
5	0.00013	0.00034	0.00088	0.00168	0.00080
11	0.00076	0.00232	0.00503	0.01047	0.00645
13	0.00013	0.00034	0.00088	0.00168	0.00081
21	0.00013	0.00034	0.00088	0.00168	0.00079

Table A.29: NAS IS, average poll point interval (milliseconds).

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chckpt size</i>	16785727		16785727		16785727		16785727		16785727	
$t_{chkpt}$	338.62	9.54	535.34	4.08	4158.10	51.57	1382.68	39.84	3427.75	73.95
$t_{restart.sparc}$	1274.49	108.85	1569.59	55.12	2126.35	45.55	1283.09	35.21	3339.40	36.40
$t_{restart.mips}$	1204.10	30.73	1534.34	28.91	2114.61	28.68	1240.38	35.50	3492.24	180.03
$t_{restart.x86}$	632.74	23.33	1054.31	11.66	3024.34	84.27	2264.12	113.57	4469.25	81.73
$t_{restart.rs/6000}$	1220.24	80.16	1529.84	18.80	2098.01	63.72	1262.62	45.77	3532.35	174.15
$t_{restart.alpha}$	624.80	9.20	1055.67	16.57	2956.11	22.49	2222.16	150.47	4454.61	70.38

Table A.30: NAS IS, time to checkpoint/restart (milliseconds).

## A.2.2 NAS EP Kernel

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
No trans	8.12	0.26	20.67	0.02	38.28	0.25	31.84	0.66	45.55	0.50
No trans, opt.	5.91	0.03	17.42	0.04	31.75	0.19	31.90	0.90	38.50	0.45
0	8.87	0.04	21.21	0.02	40.71	0.24	34.47	0.17	47.67	0.61
0, optimized	6.10	0.04	21.62	0.02	34.84	0.19	37.46	0.86	42.04	2.57
1	8.45	0.03	21.73	0.01	39.57	0.27	36.87	0.90	47.24	0.41
1, optimized	6.09	0.04	21.26	0.01	33.45	0.25	33.80	1.05	40.62	2.12
2	8.45	0.05	21.73	0.01	39.52	0.27	33.29	0.38	47.31	0.50
2, optimized	5.98	0.04	21.26	0.01	33.39	0.18	36.70	1.01	40.32	0.65
3	8.45	0.04	21.74	0.01	39.48	0.25	34.43	1.04	47.42	0.51
3, optimized	5.99	0.04	21.26	0.02	33.43	0.26	36.70	0.95	41.68	3.00
4	9.27	0.08	21.74	0.01	39.50	0.21	33.97	0.93	52.39	3.76
4, optimized	5.97	0.02	21.57	0.01	33.45	0.24	33.72	0.84	42.20	1.05
5	9.26	0.09	21.73	0.02	39.53	0.26	33.64	0.90	48.67	2.64
5, optimized	5.97	0.04	21.57	0.01	33.39	0.18	32.70	0.62	41.41	2.32
11	8.45	0.04	21.74	0.02	39.48	0.27	34.49	0.85	48.12	2.21
1, optimized	5.99	0.05	21.20	0.01	33.41	0.19	38.61	2.83	39.83	0.25
13	9.27	0.08	21.75	0.03	39.54	0.23	33.71	1.23	51.61	3.65
13, optimized	5.97	0.04	21.40	0.01	33.41	0.20	33.50	0.96	40.54	0.40
21	9.24	0.05	21.75	0.01	39.54	0.23	34.22	0.98	49.88	3.93
21, optimized	6.07	0.03	21.41	0.01	33.44	0.25	32.70	0.76	40.32	0.24

Table A.31: NAS EP, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
S <sub>ntrans</sub>	1.37		1.19		1.21		1.00		1.18	
O <sub>norm,0</sub> O <sub>opt,0</sub>	9.3%	3.2%	2.6%	24.1%	6.4%	9.7%	8.3%	17.4%	4.7%	9.2%
S <sub>trans,0</sub> E <sub>0</sub>	1.45	1.06	0.98	0.83	1.17	0.97	0.92	0.92	1.13	0.96
O <sub>norm,1</sub> O <sub>opt,1</sub>	4.0%	3.1%	5.1%	22.1%	3.4%	5.3%	15.8%	6.0%	3.7%	5.5%
S <sub>trans,1</sub> E <sub>1</sub>	1.39	1.01	1.02	0.86	1.18	0.98	1.09	1.09	1.16	0.98
O <sub>norm,2</sub> O <sub>opt,2</sub>	4.1%	1.2%	5.1%	22.1%	3.2%	5.2%	4.6%	15.0%	3.9%	4.7%
S <sub>trans,2</sub> E <sub>2</sub>	1.41	1.03	1.02	0.86	1.18	0.98	0.91	0.91	1.17	0.99
O <sub>norm,3</sub> O <sub>opt,3</sub>	4.0%	1.4%	5.2%	22.1%	3.1%	5.3%	8.1%	15.0%	4.1%	8.2%
S <sub>trans,3</sub> E <sub>3</sub>	1.41	1.03	1.02	0.86	1.18	0.98	0.94	0.94	1.14	0.96
O <sub>norm,4</sub> O <sub>opt,4</sub>	14.2%	1.0%	5.1%	23.9%	3.2%	5.3%	6.7%	5.7%	15.0%	9.6%
S <sub>trans,4</sub> E <sub>4</sub>	1.55	1.13	1.01	0.85	1.18	0.98	1.01	1.01	1.24	1.05
O <sub>norm,5</sub> O <sub>opt,5</sub>	14.0%	1.0%	5.1%	23.8%	3.3%	5.1%	5.7%	2.5%	6.9%	7.5%
S <sub>trans,5</sub> E <sub>5</sub>	1.55	1.13	1.01	0.85	1.18	0.98	1.03	1.03	1.18	0.99
O <sub>norm,11</sub> O <sub>opt,11</sub>	4.0%	1.3%	5.2%	21.7%	3.1%	5.2%	8.3%	21.0%	5.6%	3.5%
S <sub>trans,11</sub> E <sub>11</sub>	1.41	1.03	1.03	0.86	1.18	0.98	0.89	0.89	1.21	1.02
O <sub>norm,13</sub> O <sub>opt,13</sub>	14.1%	1.0%	5.2%	22.9%	3.3%	5.2%	5.9%	5.0%	13.3%	5.3%
S <sub>trans,13</sub> E <sub>13</sub>	1.55	1.13	1.02	0.86	1.18	0.98	1.01	1.01	1.27	1.08
O <sub>norm,21</sub> O <sub>opt,21</sub>	13.8%	2.8%	5.2%	22.9%	3.3%	5.3%	7.5%	2.5%	9.5%	4.7%
S <sub>trans,21</sub> E <sub>21</sub>	1.52	1.11	1.02	0.86	1.18	0.98	1.05	1.05	1.24	1.05

Table A.32: NAS EP, overhead and optimizer effectiveness.

Policy	Poll Points
0	12582925
1	4194311
2	4194312
3	4194312
4	4194332
5	4194332
11	4194312
13	4194332
21	4194332

**Table A.33: NAS EP, poll point counts.**

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	0.0005	0.0017	0.0028	0.0030	0.0033
1	0.0015	0.0051	0.0080	0.0081	0.0097
2	0.0014	0.0051	0.0080	0.0088	0.0096
3	0.0014	0.0051	0.0080	0.0088	0.0099
4	0.0014	0.0051	0.0080	0.0080	0.0101
5	0.0014	0.0051	0.0080	0.0078	0.0099
11	0.0014	0.0051	0.0080	0.0092	0.0095
13	0.0014	0.0051	0.0080	0.0080	0.0097
21	0.0014	0.0051	0.0080	0.0078	0.0096

**Table A.34: NAS EP, average poll point interval (milliseconds).**

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chckpt size</i>	314		314		314		314		314	
$t_{chckpt}$	0.37	0.47	0.76	0.10	16.26	6.92	4.24	0.22	2.43	0.03
$t_{restart.sparc}$	0.73	0.42	0.61	0.02	1.03	0.00	2.53	0.07	2.49	0.10
$t_{restart.mips}$	0.61	0.47	0.61	0.02	1.04	0.04	2.51	0.06	2.47	0.05
$t_{restart.x86}$	0.49	0.49	0.60	0.01	1.03	0.00	2.56	0.10	2.45	0.02
$t_{restart.rs/6000}$	0.61	0.47	0.61	0.03	1.04	0.05	2.55	0.04	2.43	0.02
$t_{restart.alpha}$	0.49	0.49	0.63	0.01	1.05	0.04	2.57	0.09	2.47	0.03

**Table A.35: NAS EP, time to checkpoint/restart (milliseconds).**

## A.2.3 NAS MG Kernel

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
No trans	8.29	0.05	10.13	0.05	18.65	0.05	26.08	0.72	38.79	0.75
No trans, opt.	2.87	0.03	4.44	0.05	12.28	0.18	17.46	0.40	16.33	1.22
0	8.34	0.03	10.67	0.06	19.09	0.20	27.20	0.28	39.84	1.47
0, optimized	3.00	0.08	4.91	0.03	12.87	0.07	18.37	0.30	17.90	0.71
1	8.34	0.07	10.72	0.06	18.67	0.17	27.66	0.93	39.03	1.66
1, optimized	2.90	0.06	4.90	0.03	12.34	0.20	17.78	0.21	17.36	0.77
2	8.30	0.07	10.64	0.05	18.66	0.19	26.20	0.53	38.87	0.84
2, optimized	2.90	0.05	4.87	0.05	12.27	0.03	18.90	0.71	17.01	1.42
3	8.42	0.06	10.67	0.06	18.73	0.18	27.75	2.00	38.67	1.59
3, optimized	2.91	0.08	4.86	0.03	12.33	0.19	19.14	0.50	17.09	0.93
4	8.25	0.09	10.64	0.05	18.68	0.17	26.43	0.75	40.17	1.98
4, optimized	2.89	0.06	4.92	0.05	12.27	0.05	18.37	0.14	17.38	1.09
5	8.51	0.33	10.65	0.05	18.67	0.04	28.12	2.27	39.62	1.73
5, optimized	2.90	0.05	4.87	0.03	12.35	0.25	18.11	0.66	18.83	2.35
11	8.31	0.06	10.66	0.04	18.81	0.23	26.74	0.99	38.81	1.86
1, optimized	2.91	0.08	4.93	0.05	12.43	0.21	19.34	0.45	20.14	2.26
13	8.38	0.12	10.64	0.02	18.84	0.22	27.33	1.02	38.81	1.09
13, optimized	2.91	0.07	4.92	0.04	12.38	0.06	18.05	0.28	17.98	1.23
21	8.42	0.07	10.72	0.05	19.43	0.21	25.97	0.85	40.58	2.28
21, optimized	3.05	0.06	4.97	0.03	13.18	0.19	20.58	0.77	19.30	1.14

Table A.36: NAS MG, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
S <sub>ntrans</sub>	2.89		2.28		1.52		1.49		2.37	
O <sub>norm,0</sub> O <sub>opt,0</sub>	0.6%	4.6%	5.4%	10.6%	2.4%	4.8%	4.3%	5.2%	2.7%	9.6%
S <sub>trans,0</sub> E <sub>0</sub>	2.78	0.96	2.17	0.95	1.48	0.98	1.48	0.99	2.23	0.94
O <sub>norm,1</sub> O <sub>opt,1</sub>	0.6%	1.2%	5.9%	10.4%	0.1%	0.5%	6.1%	1.8%	0.6%	6.3%
S <sub>trans,1</sub> E <sub>1</sub>	2.87	0.99	2.19	0.96	1.51	1.00	1.56	1.04	2.25	0.95
O <sub>norm,2</sub> O <sub>opt,2</sub>	0.1%	1.3%	5.1%	9.7%	0.1%	-0.1%	0.5%	8.2%	0.2%	4.1%
S <sub>trans,2</sub> E <sub>2</sub>	2.86	0.99	2.18	0.96	1.52	1.00	1.39	0.93	2.28	0.96
O <sub>norm,3</sub> O <sub>opt,3</sub>	1.5%	1.6%	5.4%	9.5%	0.4%	0.4%	6.4%	9.6%	-0.3%	4.6%
S <sub>trans,3</sub> E <sub>3</sub>	2.89	1.00	2.19	0.96	1.52	1.00	1.45	0.97	2.26	0.95
O <sub>norm,4</sub> O <sub>opt,4</sub>	-0.5%	0.7%	5.1%	10.7%	0.2%	0.0%	1.3%	5.2%	3.6%	6.4%
S <sub>trans,4</sub> E <sub>4</sub>	2.86	0.99	2.16	0.95	1.52	1.00	1.44	0.96	2.31	0.97
O <sub>norm,5</sub> O <sub>opt,5</sub>	2.6%	1.3%	5.2%	9.7%	0.1%	0.5%	7.8%	3.7%	2.1%	15.3%
S <sub>trans,5</sub> E <sub>5</sub>	2.93	1.01	2.19	0.96	1.51	1.00	1.55	1.04	2.10	0.89
O <sub>norm,11</sub> O <sub>opt,11</sub>	0.2%	1.6%	5.3%	11.0%	0.9%	1.3%	2.5%	10.8%	0.0%	23.3%
S <sub>trans,11</sub> E <sub>11</sub>	2.85	0.99	2.16	0.95	1.51	1.00	1.38	0.93	1.93	0.81
O <sub>norm,13</sub> O <sub>opt,13</sub>	1.1%	1.5%	5.1%	10.9%	1.0%	0.9%	4.8%	3.4%	0.1%	10.1%
S <sub>trans,13</sub> E <sub>13</sub>	2.88	1.00	2.16	0.95	1.52	1.00	1.51	1.01	2.16	0.91
O <sub>norm,21</sub> O <sub>opt,21</sub>	1.5%	6.6%	5.9%	11.8%	4.2%	7.3%	-0.4%	17.8%	4.6%	18.1%
S <sub>trans,21</sub> E <sub>21</sub>	2.76	0.95	2.16	0.95	1.47	0.97	1.26	0.84	2.10	0.89

Table A.37: NAS MG, overhead and optimizer effectiveness.



Policy	Poll Points
0	3858096
1	136955
2	2951
3	137575
4	11763
5	137575
11	1111343
13	1111343
21	7277331

Table A.38: NAS MG, poll point counts.

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	0.0008	0.0013	0.0033	0.0048	0.0046
1	0.0212	0.0358	0.0901	0.1298	0.1268
2	0.9840	1.6506	4.1584	6.4047	5.7643
3	0.0212	0.0353	0.0896	0.1391	0.1242
4	0.2454	0.4179	1.0435	1.5619	1.4772
5	0.0211	0.0354	0.0897	0.1317	0.1369
11	0.0026	0.0044	0.0112	0.0174	0.0181
13	0.0026	0.0044	0.0111	0.0162	0.0162
21	0.0004	0.0007	0.0018	0.0028	0.0027

Table A.39: NAS MG, average poll point interval (milliseconds).

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chckpt size</i>	7025076		7025076		7025076		7025076		7025076	
$t_{chckpt}$	414.82	20.91	550.67	3.59	2760.93	89.96	1326.48	23.53	1267.37	99.21
$t_{restart.sparc}$	482.88	6.98	596.77	8.04	1009.39	2.28	857.20	6.39	1353.10	55.16
$t_{restart.mips}$	498.54	22.37	601.02	11.21	1008.50	2.50	860.80	18.94	1368.57	54.65
$t_{restart.x86}$	302.61	17.40	516.67	4.35	1391.22	4.57	1145.69	43.85	1650.48	53.66
$t_{restart.rs/6000}$	492.43	8.68	602.51	9.95	1007.98	1.52	872.66	47.61	1328.62	13.56
$t_{restart.alpha}$	297.61	13.86	523.03	8.76	1410.89	11.00	1239.77	80.45	1672.13	44.36

Table A.40: NAS MG, time to checkpoint/restart (milliseconds).

## A.2.4 NAS CG Kernel

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
No trans	7.46	0.08	10.40	0.08	22.37	0.19	24.16	0.50	33.54	0.86
No trans, opt.	3.35	0.11	5.69	0.03	8.12	0.07	17.11	0.37	15.93	0.75
0	7.50	0.12	10.42	0.06	22.50	0.18	23.76	0.32	34.67	0.63
0, optimized	3.39	0.07	5.71	0.03	8.26	0.03	17.76	0.66	16.37	0.56
1	7.77	0.19	10.44	0.07	22.61	0.19	25.23	0.41	34.24	0.80
1, optimized	3.41	0.15	5.82	0.03	8.14	0.06	17.05	0.12	16.90	0.64
2	7.51	0.10	10.41	0.07	22.62	0.19	23.78	0.16	33.98	0.81
2, optimized	3.44	0.14	5.72	0.03	8.42	0.19	17.12	0.26	17.05	0.52
3	7.53	0.12	10.42	0.05	22.65	0.18	25.06	0.55	33.74	0.45
3, optimized	3.45	0.13	5.85	0.03	8.52	0.05	18.55	0.45	16.21	0.78
4	8.00	0.15	10.48	0.05	22.96	0.22	25.27	1.15	33.38	0.84
4, optimized	3.41	0.08	5.90	0.03	8.42	0.17	17.95	0.13	15.99	0.86
5	7.78	0.13	10.46	0.06	22.96	0.16	25.23	0.46	34.56	1.45
5, optimized	3.45	0.09	5.91	0.06	8.39	0.05	18.20	0.31	17.58	0.85
11	8.33	0.11	10.77	0.06	25.16	0.13	28.29	1.23	35.38	0.67
1, optimized	3.68	0.12	6.01	0.03	12.04	0.13	25.17	1.02	20.96	0.73
13	8.09	0.10	10.81	0.05	25.49	0.12	28.60	0.34	38.05	1.48
13, optimized	3.78	0.12	6.05	0.02	12.34	0.19	24.67	0.22	21.85	2.51
21	8.31	0.40	10.80	0.04	25.53	0.19	28.61	0.45	39.11	3.28
21, optimized	3.78	0.13	6.08	0.06	13.47	0.19	26.72	0.96	20.62	0.77

Table A.41: NAS CG, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
S <sub>ntrans</sub>	2.23		1.83		2.76		1.41		2.11	
O <sub>norm,0</sub> O <sub>opt,0</sub>	0.6%	1.1%	0.2%	0.4%	0.6%	1.8%	-1.7%	3.8%	3.4%	2.8%
S <sub>trans,0</sub> E <sub>0</sub>	2.21	0.99	1.82	1.00	2.72	0.99	1.34	0.95	2.12	1.01
O <sub>norm,1</sub> O <sub>opt,1</sub>	4.2%	1.8%	0.4%	2.3%	1.1%	0.2%	4.4%	-0.4%	2.1%	6.1%
S <sub>trans,1</sub> E <sub>1</sub>	2.28	1.02	1.79	0.98	2.78	1.01	1.48	1.05	2.03	0.96
O <sub>norm,2</sub> O <sub>opt,2</sub>	0.7%	2.6%	0.1%	0.5%	1.1%	3.7%	-1.6%	0.1%	1.3%	7.1%
S <sub>trans,2</sub> E <sub>2</sub>	2.19	0.98	1.82	1.00	2.69	0.97	1.39	0.98	1.99	0.95
O <sub>norm,3</sub> O <sub>opt,3</sub>	1.0%	2.8%	0.1%	2.8%	1.2%	4.9%	3.7%	8.4%	0.6%	1.8%
S <sub>trans,3</sub> E <sub>3</sub>	2.19	0.98	1.78	0.97	2.66	0.96	1.35	0.96	2.08	0.99
O <sub>norm,4</sub> O <sub>opt,4</sub>	7.2%	1.6%	0.8%	3.6%	2.6%	3.7%	4.6%	4.9%	-0.5%	0.4%
S <sub>trans,4</sub> E <sub>4</sub>	2.35	1.06	1.78	0.97	2.73	0.99	1.41	1.00	2.09	0.99
O <sub>norm,5</sub> O <sub>opt,5</sub>	4.3%	3.0%	0.6%	3.9%	2.6%	3.4%	4.4%	6.4%	3.0%	10.3%
S <sub>trans,5</sub> E <sub>5</sub>	2.25	1.01	1.77	0.97	2.74	0.99	1.39	0.98	1.97	0.93
O <sub>norm,11</sub> O <sub>opt,11</sub>	11.6%	9.9%	3.5%	5.7%	12.5%	48.3%	17.1%	47.1%	5.5%	31.6%
S <sub>trans,11</sub> E <sub>11</sub>	2.26	1.02	1.79	0.98	2.09	0.76	1.12	0.80	1.69	0.80
O <sub>norm,13</sub> O <sub>opt,13</sub>	8.5%	12.8%	4.0%	6.3%	13.9%	52.0%	18.4%	44.2%	13.5%	37.2%
S <sub>trans,13</sub> E <sub>13</sub>	2.14	0.96	1.79	0.98	2.07	0.75	1.16	0.82	1.74	0.83
O <sub>norm,21</sub> O <sub>opt,21</sub>	11.3%	12.9%	3.8%	6.9%	14.1%	65.9%	18.4%	56.1%	16.6%	29.4%
S <sub>trans,21</sub> E <sub>21</sub>	2.20	0.99	1.78	0.97	1.90	0.69	1.07	0.76	1.90	0.90

Table A.42: NAS CG, overhead and optimizer effectiveness.

Policy	Poll Points
0	52976
1	615785
2	283908
3	842502
4	4767261
5	4778455
11	31599790
13	35535743
21	36164967

Table A.43: NAS CG, poll point counts.

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	0.06396	0.10782	0.15596	0.33531	0.30900
1	0.00554	0.00945	0.01321	0.02768	0.02744
2	0.01211	0.02015	0.02966	0.06030	0.06006
3	0.00409	0.00694	0.01011	0.02202	0.01924
4	0.00071	0.00124	0.00177	0.00377	0.00335
5	0.00072	0.00124	0.00176	0.00381	0.00368
11	0.00012	0.00019	0.00038	0.00080	0.00066
13	0.00011	0.00017	0.00035	0.00069	0.00061
21	0.00010	0.00017	0.00037	0.00074	0.00057

Table A.44: NAS CG, average poll point interval (milliseconds).

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chkpt size</i>	960511		960511		960511		960511		960511	
$t_{chkpt}$	19.17	0.47	22.57	8.54	215.44	6.02	69.55	1.55	161.33	3.14
$t_{restart.sparc}$	62.74	2.28	84.75	2.18	132.94	14.27	77.38	4.23	173.50	8.13
$t_{restart.mips}$	62.38	2.04	86.07	1.10	118.89	0.90	74.92	1.45	153.62	6.69
$t_{restart.x86}$	33.45	1.60	63.97	1.57	168.49	1.08	119.68	5.60	225.90	9.56
$t_{restart.rs/6000}$	62.13	1.54	84.98	0.93	118.70	1.11	75.73	1.49	174.79	14.00
$t_{restart.alpha}$	31.62	1.19	63.87	2.04	167.79	1.75	117.35	4.53	218.83	14.66

Table A.45: NAS CG, time to checkpoint/restart (milliseconds).

### A.3 Environmental Simulation

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
No trans	9.93	0.06	29.60	0.04	73.35	0.36	63.52	1.05	54.40	0.22
No trans, opt.	7.11	0.03	24.93	0.05	57.21	0.14	52.80	1.23	40.43	1.44
0	12.16	0.16	31.62	0.05	80.62	0.18	75.25	1.23	59.38	0.31
0, optimized	8.21	0.08	28.17	0.06	61.52	0.16	61.48	1.72	45.48	1.33
1	11.87	0.03	32.77	0.05	81.32	0.30	72.23	1.63	59.53	0.30
1, optimized	8.51	0.03	28.15	0.06	61.30	0.24	61.45	2.76	48.04	2.20
2	12.14	0.03	32.70	0.06	81.39	0.33	72.94	1.89	58.37	0.31
2, optimized	8.36	0.08	28.29	0.05	61.85	0.26	63.54	1.73	45.40	0.59
3	12.14	0.04	32.69	0.06	81.34	0.28	72.27	1.03	58.29	0.31
3, optimized	8.38	0.08	28.28	0.06	61.89	0.29	63.73	1.99	45.59	1.37
4	11.64	0.11	31.71	0.06	82.34	0.27	75.17	1.16	59.60	0.48
4, optimized	8.41	0.05	28.30	0.05	62.06	0.27	63.29	1.35	47.34	2.91
5	11.69	0.10	31.71	0.05	82.38	0.32	74.84	1.09	58.82	1.27
5, optimized	8.42	0.07	28.28	0.06	62.08	0.24	63.58	1.73	47.79	3.85
11	11.96	0.03	32.69	0.05	81.32	0.31	73.05	0.84	58.26	0.63
1, optimized	8.46	0.06	27.75	0.05	61.84	0.24	64.70	1.50	49.30	2.55
13	11.90	0.24	31.67	0.06	81.34	0.27	74.80	1.50	60.29	0.72
13, optimized	8.63	0.04	29.12	0.05	62.15	0.36	65.44	1.81	48.76	1.22
21	11.83	0.14	31.66	0.05	81.39	0.30	75.50	2.01	59.75	0.83
21, optimized	8.61	0.14	29.14	0.06	62.04	0.28	65.07	2.51	49.14	3.90

Table A.46: LAI, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
S <sub>ntrans</sub>	1.40		1.19		1.28		1.20		1.35	
O <sub>norm,0</sub> O <sub>opt,0</sub>	22.5%	15.4%	6.8%	13.0%	9.9%	7.5%	18.5%	16.4%	9.2%	12.5%
S <sub>trans,0</sub> E <sub>0</sub>	1.48	1.06	1.12	0.95	1.31	1.02	1.22	1.02	1.31	0.97
O <sub>norm,1</sub> O <sub>opt,1</sub>	19.6%	19.7%	10.7%	12.9%	10.9%	7.2%	13.7%	16.4%	9.4%	18.8%
S <sub>trans,1</sub> E <sub>1</sub>	1.39	1.00	1.16	0.98	1.33	1.03	1.18	0.98	1.24	0.92
O <sub>norm,2</sub> O <sub>opt,2</sub>	22.3%	17.5%	10.5%	13.5%	11.0%	8.1%	14.8%	20.3%	7.3%	12.3%
S <sub>trans,2</sub> E <sub>2</sub>	1.45	1.04	1.16	0.97	1.32	1.03	1.15	0.95	1.29	0.96
O <sub>norm,3</sub> O <sub>opt,3</sub>	22.3%	17.8%	10.4%	13.5%	10.9%	8.2%	13.8%	20.7%	7.2%	12.8%
S <sub>trans,3</sub> E <sub>3</sub>	1.45	1.04	1.16	0.97	1.31	1.02	1.13	0.94	1.28	0.95
O <sub>norm,4</sub> O <sub>opt,4</sub>	17.2%	18.2%	7.1%	13.5%	12.2%	8.5%	18.3%	19.9%	9.6%	17.1%
S <sub>trans,4</sub> E <sub>4</sub>	1.38	0.99	1.12	0.94	1.33	1.03	1.19	0.99	1.26	0.94
O <sub>norm,5</sub> O <sub>opt,5</sub>	17.7%	18.4%	7.1%	13.4%	12.3%	8.5%	17.8%	20.4%	8.1%	18.2%
S <sub>trans,5</sub> E <sub>5</sub>	1.39	0.99	1.12	0.94	1.33	1.03	1.18	0.98	1.23	0.91
O <sub>norm,11</sub> O <sub>opt,11</sub>	20.4%	19.0%	10.5%	11.3%	10.9%	8.1%	15.0%	22.5%	7.1%	22.0%
S <sub>trans,11</sub> E <sub>11</sub>	1.41	1.01	1.18	0.99	1.31	1.03	1.13	0.94	1.18	0.88
O <sub>norm,13</sub> O <sub>opt,13</sub>	19.9%	21.4%	7.0%	16.8%	10.9%	8.7%	17.7%	23.9%	10.8%	20.6%
S <sub>trans,13</sub> E <sub>13</sub>	1.38	0.99	1.09	0.92	1.31	1.02	1.14	0.95	1.24	0.92
O <sub>norm,21</sub> O <sub>opt,21</sub>	19.1%	21.1%	7.0%	16.9%	11.0%	8.4%	18.8%	23.2%	9.8%	21.6%
S <sub>trans,21</sub> E <sub>21</sub>	1.37	0.98	1.09	0.92	1.31	1.02	1.16	0.96	1.22	0.90

Table A.47: LAI, overhead and optimizer effectiveness.

Policy	Poll Points
0	77428159
1	73884216
2	75624427
3	75624427
4	75637639
5	75637639
11	75686477
13	75699689
21	75699689

Table A.48: LAI, poll point counts.

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	0.000106	0.000364	0.000795	0.000794	0.000587
1	0.000115	0.000381	0.000830	0.000832	0.000650
2	0.000110	0.000374	0.000818	0.000840	0.000600
3	0.000111	0.000374	0.000818	0.000843	0.000603
4	0.000111	0.000374	0.000821	0.000837	0.000626
5	0.000111	0.000374	0.000821	0.000841	0.000632
11	0.000112	0.000367	0.000817	0.000855	0.000651
13	0.000114	0.000385	0.000821	0.000864	0.000644
21	0.000114	0.000385	0.000819	0.000860	0.000649

Table A.49: LAI, average poll point interval (milliseconds).

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chkpt size</i>	6782		6782		6782		6782		6782	
$t_{chkpt}$	1.96	0.01	2.07	0.04	22.77	1.48	11.54	0.96	6.77	0.44
$t_{restart.sparc}$	2.93	0.01	3.68	0.04	7.86	0.11	12.12	0.54	8.28	0.31
$t_{restart.mips}$	2.93	0.00	3.70	0.03	7.82	0.07	12.39	1.14	8.01	0.42
$t_{restart.x86}$	2.94	0.01	3.66	0.04	8.01	0.04	12.34	0.36	8.71	0.48
$t_{restart.rs/6000}$	2.93	0.01	3.73	0.04	7.82	0.08	12.49	0.94	7.99	0.23
$t_{restart.alpha}$	2.93	0.01	3.64	0.05	8.03	0.15	12.43	0.51	8.53	0.28

Table A.50: LAI, time to checkpoint/restart (milliseconds).

## A.4 Biological Sequence Comparison

### A.4.1 FASTA

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>No trans</i>	1.74	0.08	2.53	0.09	11.36	0.25	9.23	0.19	9.54	0.05
<i>No trans, opt.</i>	0.80	0.02	1.72	0.05	6.12	0.17	6.15	0.32	4.94	0.12
<i>0</i>	1.84	0.06	2.58	0.04	11.80	0.16	9.53	0.11	9.60	0.15
<i>0, optimized</i>	0.97	0.05	1.78	0.05	6.84	0.03	6.21	0.49	5.48	0.27
<i>1</i>	1.81	0.05	2.61	0.05	11.87	0.03	9.70	0.19	9.88	0.29
<i>1, optimized</i>	0.94	0.02	1.81	0.04	7.03	0.03	6.22	0.19	5.30	0.09
<i>2</i>	2.00	0.06	2.67	0.05	13.36	0.16	10.62	0.34	10.71	0.38
<i>2, optimized</i>	1.10	0.02	1.85	0.04	8.16	0.17	8.24	0.20	6.03	0.28
<i>3</i>	2.30	0.43	2.64	0.06	13.38	0.04	11.27	0.13	10.43	0.29
<i>3, optimized</i>	1.12	0.07	1.85	0.05	8.15	0.05	8.26	0.15	5.74	0.15
<i>4</i>	2.05	0.08	2.66	0.05	13.62	0.16	11.61	0.22	10.63	0.41
<i>4, optimized</i>	1.16	0.05	1.89	0.04	8.42	0.04	8.96	0.34	5.95	0.18
<i>5</i>	2.00	0.01	2.67	0.05	13.63	0.17	11.02	0.30	10.87	0.20
<i>5, optimized</i>	1.18	0.03	1.88	0.05	8.47	0.19	9.22	0.10	6.25	0.76
<i>11</i>	2.02	0.06	2.66	0.06	13.57	0.05	10.84	0.51	10.91	0.17
<i>1, optimized</i>	1.10	0.07	1.86	0.05	8.31	0.03	9.42	0.18	5.92	0.09
<i>13</i>	2.03	0.04	2.69	0.05	13.87	0.16	11.33	0.45	11.60	0.39
<i>13, optimized</i>	1.17	0.06	1.90	0.05	8.64	0.17	9.24	0.12	6.23	0.15
<i>21</i>	2.08	0.05	2.66	0.04	13.96	0.06	12.16	0.35	11.09	0.60
<i>21, optimized</i>	1.16	0.02	1.91	0.05	8.59	0.04	9.38	0.13	6.19	0.18

Table A.51: FASTA, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
$S_{ntrans}$	2.18		1.47		1.85		1.50		1.93	
$O_{norm,0}$ $O_{opt,0}$	5.8%	21.2%	2.1%	3.1%	3.9%	11.6%	3.2%	1.0%	0.7%	10.9%
$S_{trans,0}$ $E_0$	1.90	0.87	1.45	0.99	1.73	0.93	1.53	1.02	1.75	0.91
$O_{norm,1}$ $O_{opt,1}$	4.0%	17.5%	3.1%	4.7%	4.5%	14.7%	5.1%	1.1%	3.6%	7.3%
$S_{trans,1}$ $E_1$	1.93	0.89	1.45	0.99	1.69	0.91	1.56	1.04	1.86	0.97
$O_{norm,2}$ $O_{opt,2}$	15.0%	38.0%	5.4%	7.2%	17.6%	33.3%	15.1%	33.9%	12.3%	22.1%
$S_{trans,2}$ $E_2$	1.82	0.83	1.44	0.98	1.64	0.88	1.29	0.86	1.78	0.92
$O_{norm,3}$ $O_{opt,3}$	31.9%	40.7%	4.3%	7.0%	17.8%	33.2%	22.0%	34.2%	9.4%	16.2%
$S_{trans,3}$ $E_3$	2.04	0.94	1.43	0.98	1.64	0.88	1.36	0.91	1.82	0.94
$O_{norm,4}$ $O_{opt,4}$	17.8%	45.1%	4.9%	9.8%	19.9%	37.6%	25.7%	45.6%	11.5%	20.4%
$S_{trans,4}$ $E_4$	1.77	0.81	1.40	0.96	1.62	0.87	1.30	0.86	1.79	0.93
$O_{norm,5}$ $O_{opt,5}$	14.9%	48.1%	5.5%	9.1%	20.0%	38.4%	19.3%	49.9%	13.9%	26.5%
$S_{trans,5}$ $E_5$	1.69	0.78	1.42	0.97	1.61	0.87	1.19	0.80	1.74	0.90
$O_{norm,11}$ $O_{opt,11}$	15.8%	38.1%	5.0%	8.0%	19.5%	35.7%	17.4%	53.1%	14.4%	19.9%
$S_{trans,11}$ $E_{11}$	1.83	0.84	1.43	0.97	1.63	0.88	1.15	0.77	1.84	0.95
$O_{norm,13}$ $O_{opt,13}$	16.8%	46.5%	6.1%	10.2%	22.1%	41.1%	22.7%	50.2%	21.7%	26.2%
$S_{trans,13}$ $E_{13}$	1.74	0.80	1.41	0.96	1.60	0.86	1.23	0.82	1.86	0.96
$O_{norm,21}$ $O_{opt,21}$	19.7%	45.1%	5.1%	10.7%	22.9%	40.2%	31.7%	52.4%	16.3%	25.3%
$S_{trans,21}$ $E_{21}$	1.80	0.82	1.39	0.95	1.63	0.88	1.30	0.86	1.79	0.93

Table A.52: FASTA, overhead and optimizer effectiveness.

Policy	Poll Points
0	266559
1	1360568
2	10230798
3	10366695
4	11091474
5	11218210
11	11973635
13	12825150
21	13365830

**Table A.53: FASTA, poll point counts.**

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	0.00363	0.00667	0.02565	0.02330	0.02055
1	0.00069	0.00133	0.00516	0.00457	0.00389
2	0.00011	0.00018	0.00080	0.00081	0.00059
3	0.00011	0.00018	0.00079	0.00080	0.00055
4	0.00010	0.00017	0.00076	0.00081	0.00054
5	0.00011	0.00017	0.00076	0.00082	0.00056
11	0.00009	0.00016	0.00069	0.00079	0.00049
13	0.00009	0.00015	0.00067	0.00072	0.00049
21	0.00009	0.00014	0.00064	0.00070	0.00046

**Table A.54: FASTA, average poll point interval (milliseconds).**

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chckpt size</i>	2801497		2761417		2761417		2761417		2761417	
$t_{chckpt}$	416.75	11.77	777.18	22.39	3251.88	24.95	4210.58	93.44	3197.86	166.23
$t_{restart,sparc}$	674.99	8.59	853.77	4.76	2448.74	34.11	3649.23	91.87	2651.87	107.57
$t_{restart,mips}$	671.57	6.20	853.96	5.82	2439.23	8.46	3619.18	61.18	2762.01	74.34
$t_{restart,x86}$	619.59	7.11	826.49	5.07	2537.36	23.53	3742.57	56.60	2929.55	91.80
$t_{restart,rs/6000}$	675.25	6.26	859.25	6.56	2456.70	36.93	3783.78	133.35	2727.21	89.59
$t_{restart,alpha}$	621.38	10.46	847.75	12.17	4393.22	27.77	6506.65	285.88	3317.85	213.26

**Table A.55: FASTA, time to checkpoint/restart (milliseconds).**

## A.4.2 Smith-Waterman

Placement Policy	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
No trans	22.20	0.06	34.51	0.04	153.25	0.38	121.49	4.22	132.35	1.79
No trans, opt.	7.43	0.02	22.40	0.02	80.33	0.31	58.27	1.85	56.01	2.29
0	22.34	0.25	34.57	0.05	153.36	0.41	126.12	3.42	133.87	3.26
0, optimized	8.44	0.06	23.10	0.03	81.68	0.28	60.64	2.03	57.60	1.09
1	22.44	0.33	34.56	0.02	153.98	1.71	126.83	4.21	134.90	2.75
1, optimized	8.74	0.17	23.87	0.05	81.24	0.33	63.33	2.57	57.83	2.16
2	22.24	0.04	34.57	0.03	153.51	0.31	126.44	4.41	134.03	2.87
2, optimized	8.72	0.08	23.86	0.06	80.89	0.31	64.11	2.79	57.59	2.13
3	22.33	0.85	34.55	0.02	153.50	0.37	125.77	3.67	135.12	3.22
3, optimized	8.72	0.09	23.86	0.06	81.41	0.28	63.32	2.02	58.92	2.18
4	22.35	0.11	34.59	0.03	153.63	0.33	127.09	3.87	136.35	2.30
4, optimized	8.82	0.21	23.87	0.06	81.60	0.35	63.65	2.64	59.47	1.88
5	22.50	0.22	34.57	0.03	154.06	1.56	129.33	11.21	137.36	6.27
5, optimized	8.72	0.17	23.86	0.04	81.54	0.31	63.41	2.19	58.92	2.12
11	24.33	0.16	35.23	0.08	169.72	1.08	137.54	4.32	146.00	3.05
1, optimized	9.52	0.17	24.91	0.08	92.45	0.73	96.68	3.19	65.03	1.94
13	24.52	0.15	35.17	0.02	169.46	0.38	138.50	4.03	148.51	2.45
13, optimized	9.84	0.49	24.81	0.02	92.36	0.32	102.94	10.98	67.12	2.29
21	24.52	0.09	35.27	0.15	169.51	0.42	138.20	4.33	150.82	1.98
21, optimized	9.52	0.12	24.83	0.03	92.33	0.28	100.92	3.43	65.41	2.37

Table A.56: Smith-Waterman, execution times (seconds).

	alpha		x86		rs/6000		mips		sparc	
S <sub>ntrans</sub>	2.99		1.54		1.91		2.09		2.36	
O <sub>norm,0</sub> O <sub>opt,0</sub>	0.6%	13.6%	0.2%	3.1%	0.1%	1.7%	3.8%	4.1%	1.1%	2.8%
S <sub>trans,0</sub> E <sub>0</sub>	2.65	0.89	1.50	0.97	1.88	0.98	2.08	1.00	2.32	0.98
O <sub>norm,1</sub> O <sub>opt,1</sub>	1.1%	17.6%	0.1%	6.5%	0.5%	1.1%	4.4%	8.7%	1.9%	3.2%
S <sub>trans,1</sub> E <sub>1</sub>	2.57	0.86	1.45	0.94	1.90	0.99	2.00	0.96	2.33	0.99
O <sub>norm,2</sub> O <sub>opt,2</sub>	0.2%	17.3%	0.2%	6.5%	0.2%	0.7%	4.1%	10.0%	1.3%	2.8%
S <sub>trans,2</sub> E <sub>2</sub>	2.55	0.85	1.45	0.94	1.90	0.99	1.97	0.95	2.33	0.98
O <sub>norm,3</sub> O <sub>opt,3</sub>	0.6%	17.3%	0.1%	6.5%	0.2%	1.3%	3.5%	8.7%	2.1%	5.2%
S <sub>trans,3</sub> E <sub>3</sub>	2.56	0.86	1.45	0.94	1.89	0.99	1.99	0.95	2.29	0.97
O <sub>norm,4</sub> O <sub>opt,4</sub>	0.6%	18.7%	0.2%	6.6%	0.2%	1.6%	4.6%	9.2%	3.0%	6.2%
S <sub>trans,4</sub> E <sub>4</sub>	2.53	0.85	1.45	0.94	1.88	0.99	2.00	0.96	2.29	0.97
O <sub>norm,5</sub> O <sub>opt,5</sub>	1.3%	17.3%	0.2%	6.5%	0.5%	1.5%	6.5%	8.8%	3.8%	5.2%
S <sub>trans,5</sub> E <sub>5</sub>	2.58	0.86	1.45	0.94	1.89	0.99	2.04	0.98	2.33	0.99
O <sub>norm,11</sub> O <sub>opt,11</sub>	9.6%	28.1%	2.1%	11.2%	10.7%	15.1%	13.2%	65.9%	10.3%	16.1%
S <sub>trans,11</sub> E <sub>11</sub>	2.56	0.86	1.41	0.92	1.84	0.96	1.42	0.68	2.25	0.95
O <sub>norm,13</sub> O <sub>opt,13</sub>	10.4%	32.5%	1.9%	10.8%	10.6%	15.0%	14.0%	76.7%	12.2%	19.8%
S <sub>trans,13</sub> E <sub>13</sub>	2.49	0.83	1.42	0.92	1.83	0.96	1.35	0.65	2.21	0.94
O <sub>norm,21</sub> O <sub>opt,21</sub>	10.5%	28.2%	2.2%	10.8%	10.6%	14.9%	13.8%	73.2%	14.0%	16.8%
S <sub>trans,21</sub> E <sub>21</sub>	2.58	0.86	1.42	0.92	1.84	0.96	1.37	0.66	2.31	0.98

Table A.57: Smith-Waterman, overhead and optimizer effectiveness.



Policy	Poll Points
0	212198
1	378530
2	692001
3	699047
4	1762417
5	1769429
11	103436444
13	104506826
21	104531290

Table A.58: Smith-Waterman, poll point counts.

Placement Policy	alpha	x86	rs/6000	mips	sparc
0	0.03978	0.10886	0.38493	0.28577	0.27143
1	0.02308	0.06305	0.21462	0.16730	0.15278
2	0.01260	0.03448	0.11690	0.09265	0.08323
3	0.01247	0.03413	0.11646	0.09059	0.08428
4	0.00500	0.01355	0.04630	0.03612	0.03375
5	0.00493	0.01349	0.04608	0.03584	0.03330
11	0.00009	0.00024	0.00089	0.00093	0.00063
13	0.00009	0.00024	0.00088	0.00099	0.00064
21	0.00009	0.00024	0.00088	0.00097	0.00063

Table A.59: Smith-Waterman, average poll point interval (milliseconds).

	alpha		x86		rs/6000		mips		sparc	
	mean	std dev	mean	std dev	mean	std dev	mean	std dev	mean	std dev
<i>Chckpt size</i>	1197083		1157003		1157003		1157003		1157003	
$t_{chckpt}$	162.60	3.64	283.02	19.87	1085.98	23.20	1529.52	343.95	916.34	35.58
$t_{restart,sparc}$	242.13	7.31	329.58	3.11	969.83	40.76	1686.68	52.13	972.11	41.57
$t_{restart,mips}$	243.59	9.19	328.83	1.57	954.17	7.17	1728.45	66.34	947.75	40.43
$t_{restart,x86}$	204.59	10.13	320.06	5.48	1001.76	19.93	1893.06	157.91	1047.10	49.82
$t_{restart,rs/6000}$	239.56	4.71	331.24	1.42	959.66	26.20	1789.38	58.96	1002.62	18.98
$t_{restart,alpha}$	200.13	7.97	331.16	2.86	2792.45	24.37	5490.56	242.02	2449.60	98.51

Table A.60: Smith-Waterman, time to checkpoint/restart (milliseconds).

## References

- [1] A. Acharya, M. Ranganathan, J. Saltz, "Sumatra: A Language for Resource-aware Mobile Programs," in Vitek, J., Tschudin, C., eds. Mobile Object Systems, Springer-Verlag, 1997.
- [2] T.E. Anderson, D.E. Culler, D.A. Patterson, and the NOW team, "A Case for NOW (networks of Workstations)" *IEEE Micro*, vol. 15, no. 1, pp. 54-64, February, 1995.
- [3] R.H. Arpaci, A. Dusseau, A.M. Vahdat, L.T. Liu, T.E. Anderson, and D.A. Patterson, "The Interaction of Parallel and Sequential Workloads on a Network of Workstations," in *Proceedings of SIGMETRICS '95*, 1995.
- [4] Y. Artsy and R. Finkel, "Designing a Process Migration Facility: The Charlotte Experience," *IEEE Computer*, pp. 47-56, September, 1989.
- [5] D. Bailey, J. Barton, T. Lasinski, and H. Simon, et. al., "The NAS Parallel Benchmarks," Technical Report RNR-94-007, NASA Ames, Moffett Field, California, March 1994.
- [6] A. Barak, G. Shai, and R.G. Wheeler, The MOSIX Distributed Operating System: Load Balancing for Unix, Springer-Verlag, Berlin, 1993.
- [7] A. Beguelin, E. Seligman, and M. Starkey, "Dome: Distributed Object Migration Environment," Carnegie Mellon University Technical Report CMU-CS-94-153, May 1994.
- [8] 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.
- [9] M. Bishop and M. Valence, "Process Migration for Heterogeneous Distributed Systems," Dartmouth College Technical Report PCS-TR95-264, August 21, 1995.
- [10] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 65-76, January, 1987.
- [11] 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.
- [12] L. Cardelli, "Oblique: A Language with Distributed Scope," Technical Report, Digital Equipment Corporation, May 1995.
- [13] N. Carriero, D. Gelernter, T.G. Mattson, and A.H. Sherman, "The Linda Alternative to Message-passing Systems," *Parallel Computing*, vol. 20, pp. 633-655, 1994.
- [14] J. Casas, D.L. Clark, R. Konuru, S.W. Otto, R.M. Prouty, J. Walpole, "Adaptive Load

- Migration Systems for PVM,” in *Proceedings of Supercomputing '94*, pp. 390-399, November, 1994.
- [15] J. Casas, D.L. Clark, R. Konuru, S.W. Otto, R.M. Prouty, J. Walpole, “MPVM: A Migration Transparent Version of PVM,” Technical Report, Oregon Graduate Institute of Science and Technology, Portland, Oregon, February, 1995.
- [16] 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.
- [17] H. Cejtin, S. Jagannathan, and R. Kelsey, “Higher-Order Distributed Objects,” *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 5, pp. 704-739, September, 1995.
- [18] K.M. Chandy and L. Lamport, “Distributed Snapshots: Determining Global States of Distributed Systems,” *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63-75, February, 1985.
- [19] C. Cowan, H.L. Lutfiyya, and M.A. Bauer, “Performance Benefits of Optimistic Programming: A Measure of HOPE,” in *Proceedings of the 4th IEEE Symposium on High Performance Distributed Computing*, pp. 197-204, 1995.
- [20] F. Douglass and J. Osterhout, “Process Migration in the Sprite Operating System,” in *Proceedings of the 7th International Conference on Distributed Computing*, pp. 18-25, 1987.
- [21] 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, February, 1989.
- [22] D.L. Eager, E.D. Lazowska, and J. Zahorjan, “Adaptive Load Sharing in Homogeneous Distributed Systems,” *IEEE Transactions on Software Engineering*, vol. SE-12, no. 5, pp. 662-675, May 1986.
- [23] D.L. Eager, E.D. Lazowska, and J. Zahorjan, “The Limited Performance Benefits from Migrating Active Processes for Load Sharing”, *ACM SIGMETRICS*, pp. 662-675, May 1988.
- [24] E.N. Elnozahy, D.B. Johnson, Y.M. Wang, “A Survey of Rollback-Recovery Protocols in Message-Passing Systems,” Technical Report CMU-CS-96-181, Carnegie Mellon University, October, 1996.
- [25] M.M. Eshaghian, “An Introduction to Heterogeneous Computing,” in *Heterogeneous Computing*, M.M. Eshaghian, ed., Artech House Publishers, pp. 1-16, 1996.

- [26] M.R. Eskicioglu, "Design Issues of Process Migration Facilities in Distributed Systems," *IEEE Technical Committee on Operating Systems Newsletter*, vol. 4, no. 2, pp. 3-13, Winter, 1989.
- [27] S.I. Feldman, D.M. Gay, M.W. Maimone, and N.L. Schryer, "A Fortran-to-C Converter," Computing Science Technical Report no. 149, AT&T Bell Laboratories, 1990.
- [28] A.J. Ferrari and V.S. Sunderam, "Multiparadigm Distributed Computing with TPVM," *Journal of Concurrency, Practice and Experience*, (to appear).
- [29] I. Foster, C. Kesselman, S. Tuecke, "The Nexus Task-parallel Runtime System," In *Proceedings of the 1st International Workshop on Parallel Processing*, 1994.
- [30] I. Foster, J. Geisler, W. Nickless, W. Smith, S. Tuecke, "Software Infrastructure for the I-WAY High-Performance Distributed Computing Experiment," in *Proceedings of the 5th IEEE Symposium on High Performance Distributed Computing*, pp. 562-570, 1996.
- [31] I. Foster, C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit", *International Journal of Supercomputing Applications* (to appear).
- [32] R.F. Freund and D. S. Cornwell, "Superconcurrency: A Form of Distributed Heterogeneous Supercomputing," *Supercomputing Review*, vol. 3, pp. 47-50, October, 1990.
- [33] R.F. Freund and H.J. Siegel, "Heterogeneous Processing," *IEEE Computer*, vol. 26, no. 6, pp. 13-17, June, 1993.
- [34] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam, PVM: Parallel Virtual Machine, MIT Press, 1994.
- [35] J. Gosling and H. McGilton, "The Java Language Environment: a White Paper," Sun Microsystems, Mountain View, CA, October, 1995.
- [36] R. Gray, G. Cybenko, D. Kotz, and D. Rus, "Agent TCL," in Cockayne, W., Zypa, M., eds. Itinerant Agents: Explanations and Examples with CDROM, Manning Publishing, 1997.
- [37] A.S. Grimshaw, "Easy-to-Use Object-Oriented Parallel Processing with Mentat," *IEEE Computer*, vol. 26, no. 5, pp. 39-51, 1993.
- [38] 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.
- [39] A.S. Grimshaw, W.A. Wulf, and the Legion team, "The Legion Vision of a Worldwide Virtual Computer," *Communications of the ACM*, vol. 40, no. 1, January, 1997.
- [40] A.S. Grimshaw, A. Nguyen-Tuong, M.J. Lewis, and M. Hyett, "Campus-Wide Comput-

- ing: Results Using a Legion Prototype at the University of Virginia,” *International Journal of Supercomputing Applications*, (to appear).
- [41] W. Gropp, E. Lusk, and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, 1994.
- [42] M. Harchol-Balter and A.B. Downey, “Exploiting Process Lifetime Distributions for Dynamic Load Balancing,” Technical Report UCB/CSD-95-887, University of California at Berkely, November, 1995.
- [43] D.R. Jefferson, “Virtual Time”, *ACM Transaction on Programming Languages and Systems*, vol. 7, no. 3, pp.404-425, July 1985.
- [44] D. Johansen, N.P. Sudmann, and R. van Renesse, “Performance issues in TACOMA,” in *Third Workshop on Mobile Object Systems, 11th European Conference on Object-Oriented Programming*, Jyvaskyla, Finland, June 9-13, 1997.
- [45] D. Johansen, R. van Renesse, and F. Schneider, “An Introduction to the TACOMA Distributed System, Version 1.0,” Computer Science Technical Report 95-23, University of Tromsø, Tromsø, Norway, June, 1995.
- [46] B.W. Kernighan and D.M. Ritchie, The C Programming Language, Second Edition, Prentice Hall, 1988.
- [47] A.A. Khokhar, V.K. Prasanna, M.E. Shaaban, and C. Wang, “Heterogeneous Computing: Challenges and Opportunities,” *IEEE Computer*, vol. 26, no. 6, pp. 18-27, June, 1993.
- [48] F.C. Knabe, “Language Support for Mobile Agents,” PhD Thesis, School of Computer Science, Carnegie Mellon University, available as Technical Report CMU-CS-95-223, December, 1995.
- [49] P. Krueger and M. Livny, “A Comparison of Preemptive and Non-Preemptive Load Distributing,” in *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 123-130, June, 1988.
- [50] 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.
- [51] M.J. Lewis and R.E. Cline, “PVM Communication Performance in a Switched FDDI Heterogeneous Distributed Computing Environment,” in *Proceedings of the IEEE Workshop on Advances in Parallel and Distributed Systems*, Princeton, N.J., October, 1993.
- [52] M.J. Lewis and A.S. Grimshaw, “The Core Legion Object Model,” in *Proceedings of*

- IEEE High Performance Distributed Computing 5*, pp. 551-561, Syracuse, NY, August 6-9, 1996.
- [53] C.C. Li and W.K. Fuchs, "CATCH: Compiler-assisted Techniques for Checkpointing," in *Proceedings of the 20th International Symposium on Fault Tolerant Computing*, pp. 74-81, 1990.
- [54] M.J. Litzkow, M. Livny, and M.W. Mutka, "Condor—A Hunter of Idle Workstations," in *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pp. 104-111, 1988.
- [55] M.J. Litzkow and M. Solomon, "Supporting Checkpointing and Process Migration Outside the UNIX Kernel," in *Proceedings of USENIX*, pp. 283-290, January, 1992.
- [56] G. Maguire and J. Smith, "Process Migrations: Effects on Scientific Computation," *ACM SIGPLAN*, vol. 23, no. 2, pp. 102-106, March, 1988.
- [57] K. Mandelberg and V.S. Sunderam, "Process Migration in Unix Networks," in *Proceedings of the USENIX Winter Conference*, pp. 357-363, 1988.
- [58] F. Mattern, "Virtual Time and Global States of Distributed Systems," in *Proceedings of the Workshop on Parallel and Distributed Algorithms*, Chateau de Bonas, France, pp. 215-226, October, 1988.
- [59] F. Mattern, "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation," *Journal of Parallel and Distributed Computing*, vol. 18, pp. 423-434, 1993.
- [60] C.R. Mechoso, J.D. Farrara, and J.A. Spahr, "Running a Climate Model in a Heterogeneous Distributed Computer Environment," in *Proceedings of the 3rd IEEE Symposium on High Performance Distributed Computing*, pp. 79-84, April, 1994.
- [61] Message Passing Interface Forum, "MPI-2: Extensions to the Message Passing Interface," July 18, 1997.
- [62] D.S. Milojicic, F. Douglass, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration," TOG RI Technical Report, 1996.
- [63] R. Mirchandaney, D. Towsley, and J.A. Stankovic, "Adaptive Load Sharing in Heterogeneous Distributed Systems," *Journal of Parallel and Distributed Computing*, vol. 9, pp. 331-346, 1990.
- [64] M. Nuttall, "Survey of Systems Providing Process or Object Migration," Imperial College Research Report DoC 94/10, May, 1994.
- [65] J.K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

- [66] W.R. Pearson and D. Lipman, "Improved Tools for Biological Sequence Analysis," *Proc. National Academy of Science USA*, vol. 85, pp. 2444-2448, 1988.
- [67] H. Peine, "An introduction to mobile agent programming and the Ara system," ZRI-Report 1/97, Department of Computer Science, University of Kaiserslautern, Germany, 1997.
- [68] H. Peine and T. Stolpmann, "The Architecture of the Ara Platform for Mobile Agents," in Rothermel, K., Popescu-Zeletin, R., eds. *Proceedings of the First International Workshop on Mobile Agents: MA'97*, Berlin, Germany, April 7-8, 1997. Lecture Notes in Computer Science no. 1219, Springer Verlag, 1997.
- [69] J.S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix," in *Proceedings of USENIX Winter 1995 Technical Conference*, New Orleans, LA, January 16-20, 1995.
- [70] M.L. Powell and B.P. Miller, "Process Migration in DEMOS/MP," in *Proceedings of the Ninth Symposium on Operating Systems Principles in ACM Operating Systems Review*, vol. 17, no. 5, pp. 110-118, 1983.
- [71] A. Reinefeld, R. Baraglia, T. Decker, J. Gehring, D. Laforenza, F. Ramme, T. Rvmke, and J. Simon, "The MOL Project: An Open Extensible Metacomputer," in *Proceedings of the Heterogenous Computing Workshop, HCW97*, IEEE Computer Society Press, pp. 17-31, 1997.
- [72] J. Robinson, S.H. Russ, B. Flachs, and B. Heckel, "A Task Migration Implementation for the Message Passing Interface," in *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Systems*, Syracuse, NY, August, 1995.
- [73] N.G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *IEEE Computer*, vol. 25, no. 12, pp. 33-44, December, 1992.
- [74] H.J. Siegel, H.G. Dietz, and J.K. Antonio, "Software Support for Heterogeneous Computing," in A.B. Tucker, ed., *The Computer Science and Engineering Handbook*, CRC Press, pp. 1886-1913, 1997.
- [75] L. Smarr and C.E. Catlett, "Metacomputing," *Communications of the ACM*, vol. 35, no. 6, pp. 45-52, 1992.
- [76] J.M. Smith, "A Survey of Process Migration Mechanisms," *Operating Systems Review*, vol. 22, no. 3, pp. 28-40, July, 1988.
- [77] P. Smith and N.C. Hutchinson, "Heterogeneous Process Migration: The Tui System," Technical Report, University of British Columbia, February 28, 1996.

- [78] T.F. Smith and M.S. Waterman, "Identification of Common Molecular Subsequences," *Journal of Molecular Biology*, vol. 147, pp. 195-197, 1981.
- [79] Sun Microsystems, *External Data Representation Reference Manual*, Sun Microsystems, January, 1985.
- [80] Sun Microsystems, *Java Object Serialization Specification*, Revision 0.9, 1996.
- [81] V.S. Sunderam, "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2, no. 4, pp. 315-339, December, 1990.
- [82] B. Steensgaard and E. Jul, "Object and Native Code Thread Mobility Among Heterogeneous Computers," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, December, 1995.
- [83] G. Stellner, "Consistent Checkpoints of PVM Applications," in *Proceedings of the First European PVM Users Group Meeting*, 1994.
- [84] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI," Technical Report, Institut für Informatik der Technischen Universität München, 1996.
- [85] V. Strumpfen and B. Ramkumar, "Portable Checkpointing and Recovery in Heterogeneous Environments," Technical Report, Department of Electrical and Computer Engineering, University of Iowa, 1996.
- [86] M.M. Theimer, K.A. Lantz, and D.R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," in *Proceedings of the Tenth ACM Symposium on Operating System Principles*, December 1985.
- [87] M.M. Theimer, and B. Hayes, "Heterogeneous Process Migration by Recompile," in *Proceedings of the 11th International Conference on Distributed Computing Systems*, Arlington, TX, pp. 18-25, May 1991.
- [88] L.H. Turcotte, "A Survey of Software Environments for Exploiting Networked Computing Resources," Technical Report, Engineering Research Center for Computational Field Simulation, Mississippi State, MS, June, 1993.
- [89] S. Venkatesan and T. Juang, "Efficient Algorithms for Optimistic Crash Recovery," *Distributed Computing*, vol. 8, no. 2, pp. 105-114, 1994.
- [90] 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, November, 1994.
- [91] Y. Wang and R.J.T. Morris, "Load Sharing in Distributed Systems," *IEEE Transactions on Computers*, vol. C-94, no. 3, pp. 204-217, March, 1985.



- [92] J. White, "Mobile Agents White Paper," General Magic, <http://www.genmagic.com/agents/Whitepaper/whitepaper.html>, 1996.
- [93] S. White, A. Ålund, and V.S. Sunderam, "Performance of the NAS Parallel Benchmarks on PVM Based Networks," *Journal of Parallel and Distributed Computing*, vol. 26, no. 1, pp. 61-71, April 1995.
- [94] F.I. Woodward, T.M. Smith, and W.R. Emanuel, "A Global Land Primary Productivity and Phytogeography Model," *Global Biogeochemical Cycles*, vol. 9, no. 4, pp. 471-490, December, 1995.
- [95] E.R. Zayas, "Attacking the Process Migration Bottleneck," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pp. 13-24, 1987.
- [96] H. Zhou and A. Geist "Receiver Makes Right Data Conversion in PVM," in *Proceedings of 14th International Conference on Computers and Communications*, pp. 458-464, March 1995.
- [97] S. Zhou, J. Wang, X. Zheng, and P. Delisle, "Utopia: A Load-sharing Facility for Large Heterogeneous Distributed Computing Systems," *Software - Practice and Experience*, vol. 23, no. 2, pp. 1305-1336, December, 1993.