Autopilot: Adaptive Control of Distributed Applications*

Randy L. Ribler Jeffrey S. Vetter Huseyin Simitci
Daniel A. Reed
{ribler,jsv,simitci,reed}@cs.uiuc.edu

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

Abstract

With increasing development of applications for heterogeneous, distributed computing grids, the focus of performance analysis has shifted from a posteriori *optimization on homogeneous parallel systems to application tuning for heterogeneous resources with time varying availability. This shift has profound implications for performance instrumentation and analysis techniques.* Autopilot *is a new infrastructure for dynamic performance tuning of heterogeneous computational grids based on closed loop control. This paper describes the* Autopilot *model of distributed sensors, actuators, and decision procedures, reports preliminary performance benchmarks, and presents a case study in which the* Autopilot *library is utilized in the development of an adaptive parallel input/output system.*

1. Introduction

Although both programming models and parallel computer systems continue to evolve rapidly, most performance analysis remains grounded in a process developed over forty years ago:

- *Application instrumentation.* Application code may be instrumented automatically (e.g., by object code patching or by compilers) or manually by inserting calls to instrumentation library routines. During subsequent execution, the instrumentation library records pertinent performance data, including procedure, loop, and basic block execution counts and times.
- *Performance data extraction.* After instrumentation, performance data is captured from one or more program executions. Ideally, these executions involve input data and computing resources typical of those encountered in a production environment.
- *Analysis and visualization.* After post-processing, performance data is visualized and analyzed to identify application program performance bottlenecks (e.g., using text-based profiling tools or visualization systems like IPS-2 [8], AIMS [15], or Pablo [9]).
- *Application optimization.* Based on measurement and analysis, either the program is modified to alleviate the perceived bottlenecks or runtime system policies are adjusted to better match program resource requests.

Although effective for application codes with repeatable behavior, this *a posteriori* tuning model is ill-suited to complex, multidisciplinary applications with time varying resource demands that execute on heterogeneous collections of geographically distributed computing resources. Not only may the execution context not be repeatable *across* program executions, resource availability may change *during* execution.

Consequently, we believe performance tuning systems must evolve to reflect changing requirements, applying real-time adaptive control techniques to dynamically adapt to changing application resource demands and system resource availability. Using real-time performance data, these adaptive systems can either be

*This work was supported in part by the Defense Advanced Research Projects Agency under DARPA contracts DABT63-94-C0049 (SIO Initiative), F30602-96-C-0161, and DABT63-96-C-0027 by the National Science Foundation under grants NSF CDA 94-01124 and ASC 97-20202, and by the Department of Energy under contracts DOE B-341494, W-7405-ENG-48, and 1-B-333164.

steered interactively or by an intelligent decision support system. In this model, the goal of the performance analyst is to formulate general-purpose performance optimization rules that can be applied by the adaptive infrastructure.

Based on this hypothesis, the remainder of this paper is organized as follows. In §2, we briefly describe related work in adaptive control, followed in §3 by a description of the *Autopilot* toolkit for closed loop adaptive performance tuning and resource management. In §4, we report the results of initial performance experiments with *Autopilot*, followed in §5 by a summary of experiences with PPFS II, our first major application of *Autopilot*. Finally, §6 briefly describes our research plans.

2. Related Work

Software mediated dynamic adaptation has been applied in many domains, including real-time and fault-tolerant systems, dynamic load balancing, on-line configuration management [4] and adaptive input/output systems. The *Autopilot* toolkit differs by emphasizing portable performance steering and closed loop adaptive control and by decoupling the steering infrastructure from the policy domain.

Likewise, interactive application steering [14] has a long and rich history, particularly in the context of scientific and immersive visualization. By separating performance measurement, control and decision making, *Autopilot* enables system designers to replace software decision procedures with real-time visualization and interactive steering [11] when the rate of change admits human control.

Finally, a plethora of techniques for distributed decision making have been proposed, ranging from decision tables and trees through standard control theory to fuzzy logic. Although each has strengths, fuzzy logic targets precisely the attributes of the performance optimization problem that challenge classic techniques [16], namely conflicting goals and poorly understood optimization spaces. *Autopilot* builds on this observation by coupling a configurable fuzzy logic rule base for distributed decision making with wide area performance sensors and policy control actuators.

3. *Autopilot* Software Components

Any adaptive control system must implicitly or explicitly monitor pertinent system state(s), determine what changes are needed, and realize those changes to meet the desired goals. To dynamically optimize application and runtime system behavior for distributed, computational grids, a closed loop adaptive performance system must include some variant of the following:

- *Distributed performance sensors* that can capture quantitative application and system performance data and generate both qualitative descriptions of resource demands and quantitative performance metrics.
- *Software actuators* that can enable and configure application behavior and resource management policies.
- *Decision procedures*, both local (e.g., per parallel task) and global (e.g., per parallel program), for selecting resource management policies and enabling actuators based on observed application resource requests and the system responses captured by performance sensors.
- *Distributed name servers* that support registration by remote sensors and actuators and property-based requests for sensors and actuators by remote clients.
- *Sensor and actuator clients* that interact with remote sensors and actuators, monitoring sensor data and issuing commands to actuators.
- *Robust decision mechanisms* that exploit data from distributed sensors to balance often conflicting optimization goals.

Below, we describe each of the *Autopilot* components and their design rationale in greater detail.

3.1 Nexus Toolkit

Based on the successful Nexus [3] communication substrate, the *Autopilot* toolkit embodies sensors, actuators, decision procedures, servers, and clients in a policy and platform independent infrastructure. Nexus creates a global address space that encompasses all processes executing on a network.

The Nexus term *endpoint* refers to an address in the global address space, and the term *startpoint* identifies a pointer to an endpoint. In addition to a global address, Nexus endpoints specify a set of message handlers that are invoked when messages are sent to the endpoint.

Before a client can communicate with a sensor or actuator, it must obtain a startpoint to that sensor or actuator. Similarly, sensors and actuators must obtain startpoints to their clients. The *Autopilot* manager, described in §3.4, is a daemon process that acts as a

name server, providing remote clients with the ability to obtain startpoints to sensors and actuators.

In the following sections we describe each of the *Autopilot* components and their design rationale in greater detail.

3.2. Sensors

Sensors extract qualitative and quantitative performance data from executing applications, providing the requisite data for informed decision making. Because measurement perturbs the system under study, any sensor implementation must minimize the overhead for data capture and extraction. Moreover, in the distributed collection of sequential and parallel systems forming a computational grid [2], some subset of the sensor data must be extracted and transmitted to remote sites for global decision making. Finally, to avoid oscillating decisions due to stale sensor data, the *lag* between data collection and processing must be small.

3.2.1 Sensor Design Principles

The ideal sensor is lightweight, minimally perturbing, remotely accessible, and applicable to multiple architectures and programming models. Approaching this ideal is possible only if the sensor implementation can be tailored to the execution environment, trading computation against communication based on available resources and acceptable perturbation.

For example, sensors can either capture and transmit raw data (e.g., a stream of file request sizes) or compute and periodically transmit derived metrics (e.g., a sliding window average of request sizes). Unless the sensor and the data sink are co-located or the metric computation is extraordinarily expensive (e.g., a critical path calculation), calculating metric values at the sensors is usually less invasive. Because appropriate choices depend on the execution context, application perturbation constraints, and metric complexity, any sensor implementation should include configuration options for adjusting the balance.

If the raw sensor data stream is monitored, transmission overhead can be reduced by data buffering, albeit at the expense of increased lag. Hence, sensors should also provide controls for adjusting buffer sizes and transmission frequencies.

As distributed resource availability and application demands change, the requisite performance data change, necessitating deactivation of current sensors and activation of new ones. To dynamically change the mix of performance data without detailed knowledge of the physical location of all software components, remote clients should be able to locate sensors via descriptions of their properties (e.g., TCP or I/O sensors). Concomitantly, the sensor infrastructure should support sensor registration and dynamic activation.

3.2.2 Autopilot Sensor Features

Based on the design principles just outlined, *Autopilot* implements a suite of configurable sensors that can be dynamically activated, managed and deactivated.

Property Lists. Every sensor has a set of associated properties that are defined at the time the sensor is created. These typically include the sensor name, type, identifier, network IP address and any user-defined attribute-value pairs. Remote clients can specify property lists during queries to the *Autopilot* manager. The manager then provides the client with start points to all remote sensors that currently satisfy the query. Additionally, clients can request notification each time a sensor with the specified set of properties is created. Together, these capabilities allow clients to acquire and manage remote sensors without embedded knowledge of their physical location or creation times.

Activation Modes. Sensor creation associates a set of application or runtime system variables with the sensor monitoring software. After activation by a remote client, the sensor periodically records the values of these variables and optionally applies a set of real-time reductions to the values before transmission to remote sites.

Sensors collect data in either threaded or non-threaded modes. In the threaded mode, a monitoring thread records the values of the associated variables at intervals specified either during sensor creation or by a remote client. The nonthreaded monitoring mode relies on insertion of sensor monitoring calls in either source or object application or library code.

Data Reduction. For local data reduction, sensors can apply data transformation functions to captured data prior to recording. These *attached functions* accept raw sensor data as input, and record function output for transmission.

Attached functions can compute simple statistics (e.g., sliding window averages) or more complex transformations. For example, one set of attached functions generates qualitative file access pattern descriptions from input/output request measures (e.g., converting a sequence of file offsets to sequential, strided, or random access descriptions).

```
// Define Properties for RequestSize Sensor
ApProperties RequestSizeProperties(progName, mgrName);
RequestSizeProperties.addProperty("Name",
  "RequestSizeSensor");
RequestSizeProperties.addProperty("Application",
  "PPFS II");

// Construct RequestSizeSensor.
ApIntegerSensor RequestSizeSensor("RequestSizeSensor",
                       RequestSizeProperties,
                       requestSize,
                       variableCount=1, bufferSize=8);

// Register Sensor with Autopilot Manager
RequestSizeSensor.registerStartPoint();
```

Figure 1. Sensor registration code.

3.2.3 Sensor Creation Example

To illustrate the properties of *A*utopilot sensors, Figure 1 shows the creation of a sensor named `RequestSizeSensor` using C++ syntax. This sensor is used to monitor the application-level variable `requestSize`. The `ApProperties` constructor specifies the program name and the name of the host executing the *Autopilot* Manager. The `addProperty` method then assigns properties to the sensor as key-value pairs. Because this sensor is monitoring an integer, the `ApIntegerSensor` sensor variant is used. *Autopilot* supports scalar and array sensors for all the basic data types, as well as a *MultiSensor* that can monitor aggregates of variables of different data types.

The parameters for the `ApIntegerSensor` specify the sensor name, the sensor properties, the memory location that will be monitored, the number of integers that comprise the monitored location, and the size of the buffer that should be used to hold sampled values prior to their transmission to clients. In this case, up to eight samples of the scalar integer `requestSize` may be buffered.

Finally, the `registerStartPoint` method completes the sensor production process with the transmission of the sensor startpoint and property list to the *Autopilot* Manager. Later, the sensor destructor will inform the *Autopilot* manager of the sensor's demise.

3.3. Software Actuators

Autopilot actuators allow clients to modify the values of application variables and to remotely invoke application level functions. Typically, actuators are used to modify parameter values or to change resource management policies (e.g., changing file caching policies).

To simplify management and code development, *Autopilot* actuators share most of the features of sensors, including property lists, attached functions, and dynamic insertion and control. Hence, actuators can be identified using a set of properties, are managed by *Autopilot* managers, and can be inserted or deleted dynamically.

Actuator creation follows the same basic model used for sensor creation demonstrated in Figure 1. Finally, actuator attached functions can mediate remote command manipulations in the context of local data (e.g., bounding a remote request to increase cache size based on local memory availability).

3.4. Distributed Name Servers

As should now be clear, an *Autopilot* manager coordinates connections between sensors, actuators, and remote clients. Because the managers function as sensor/actuator name servers, clients can acquire remote sensors or actuators without knowledge of their physical or logical location. This generality allows client objects to dynamically attach to geographically distributed software components, exercise control, then relinquish the attachment.

Sensors and actuators register their properties and Nexus startpoints with an *Autopilot* manager immediately after creation, and inform the manager when they are destroyed. Clients specify a set of desired properties, and the manager provides startpoints to the sensors or actuators that match the request. This allows the client to establish direct communication with application instrumentation.

3.5. Remote Clients

Remote clients exploit sensors and actuators to realize distributed control. As described previously, clients connect to remote sensors and actuators using startpoints obtained from an *Autopilot* manager. After connection, sensors send data to all the connected remote clients, where data receipt activates a client-specified callback to process the data. Clients can also change sensor behavior, modifying attributes such as activation, buffer size, and sampling rate.

3.6. Flexible Decision Mechanisms

Sensors provide the requisite data for decision making, and actuators implement decisions — distributed decision making is the final component of closed loop adaptive control. Although one can implement decision

procedures using algorithmic or decision table techniques, our experience with parallel resource management policies [11, 6] suggests that more flexible mechanisms are needed to accommodate complex, poorly understood policy spaces. Simply put, constructing decision tables presumes a deep understanding of the resource optimization space and the relation of system controls to locations in that space.

In contrast to classic decision procedure techniques and their emphasis on consistent parameter space division, fuzzy logic allows one to elegantly balance potentially conflicting goals (e.g., minimize response time and maximize throughput). Moreover, by changing the fuzzy logic rule base, one can adjust the control system or even retarget it to a new domain without extensive software development. *Autopilot* includes a fuzzy logic engine that accepts sensor inputs, fuzzifies the values for rule application, computes the relative truth of each rule, and defuzzifies the rule consequents to activate remote actuators.

4. Autopilot Performance

The performance of an adaptive steering system like *Autopilot* depends on the interplay of a great many variables and configuration options, including sensor buffer sizes, threaded and non-threaded sensor modes, sensor fan out (number of clients/sensor), dynamic data reduction via attached functions, fuzzy logic rule base complexity, proximity of sensors/actuators and their clients, available network bandwidth, data throttling via sensor enablement/disablement, and actuator synchronization. We briefly discuss the implications of two of the most important of these issues. Following this, we report the results of an initial performance study and illustrate *Autopilot*'s use in an adaptive parallel file system.

4.1. Data Buffering and Data Reduction

As described in §3.2, when an *Autopilot* sensor records data, it can be buffered as raw data or first processed by an attached function. In either case, the choice of buffer size determines transmission frequency and overhead, as well as the buffering latency experienced by data before transmission. Small buffers reduce the latency before transmission, albeit by increasing transmission frequency and overhead.

In the absence of constraints on acceptable data buffering latency, the transmission costs for a given communication substrate determine lower bounds on efficient buffer sizes. For sizes below this point, communication latency will dominate total transmission overhead.

Attached functions can reduce data transmission costs by aggregating a sequence of raw sensor values (e.g., by computing means, minima, or maxima). However, for a fixed buffer size, this aggregation increases the buffering latency prior to transmission. For this reason, *Autopilot* attached functions can force immediate data transmission if necessary to bound buffering latency.

4.2. Performance Evaluation

As just described, an optimal choice of performance buffer size depends on both the parameters of the communication substrate and the acceptable buffering latency prior to transmission. To identify effective operating points, we conducted a series of experiments in three contexts: geographic area, local area, and intra-system (interprocessor) control. In all three cases, we measured the round trip delay to send data from a sensor to a remote client, and then back to the sensor process.

4.2.1 Experimental Testbed

As a basis for performance analysis, we conducted experiments on a Sun Ultra 1 Model 170 (170 MHz UltraSPARC processor) with 64 MB of memory, and an SGI Origin2000 with 32 195MHz R10000 processors and 4 GB of memory.

4.2.2 Communication Overhead

Our experiments showed that the local cost for sensor monitoring of a single variable on the Sun Ultra was a modest four microseconds. Hence, *Autopilot* sensor overhead is comparable to other state-of-the-art measurement systems like Paradyn [7], making it possible to conduct fine-grained measurements.

Figure 2 shows the round trip delay for data buffering and transmission in the geographic, local area, and intra-system contexts as a function of the sensor buffer size. In four of these cases the underlying Nexus communication was based on TCP, in one case it was based on MPI, and in the final case it was based on shared memory.

As expected, for small buffer sizes, communication overhead is dominated by communication latency, ranging from roughly one millisecond between processes on a single system to forty milliseconds in the wide area. For these transmissions, there is little advantage to using buffer sizes smaller than 4 KB unless

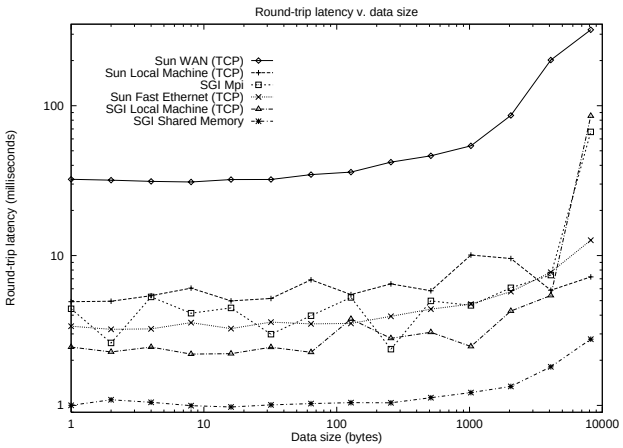


Figure 2. Round Trip Communication Latency

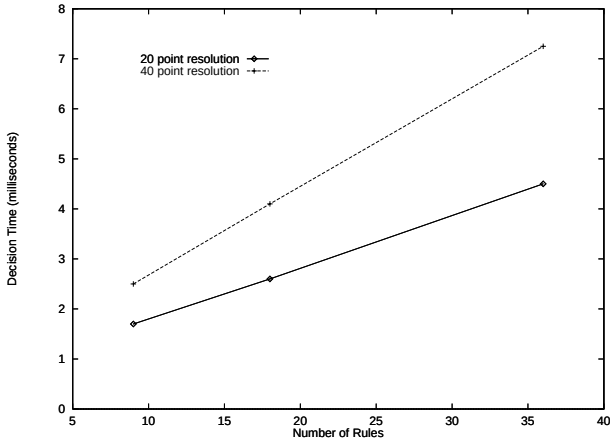


Figure 3. Fuzzy Rule Evaluation Overhead

sensor data buffering delays mandate more frequent transmission.

Interestingly, communication latency on a single Sun Ultra is longer than the latency between two systems connected with fast Ethernet. As Table 1 suggests, the primary cause for this seeming disparity is context switch overhead, system traps, and page faults.

Measure	Local	Fast Ethernet
Minor page faults	1916	100
System traps	101417	35755
Context switches	42794	22942

Table 1. Relative System Overhead

Because Nexus supports a variety of wide area communication protocols (e.g., native ATM, UDP, and TCP), as well as the ability to choose specialized low latency protocols for specific contexts (e.g., shared memory or MPI), one can dynamically configure *Autopilot* communication to minimize communication costs.[1]

4.2.3 Decision Procedure Overhead

Finally, to assess the overhead for fuzzy logic control, we measured the time needed to evaluate a rule base as a function of both the number of rules and the resolution of the fuzzy sets. The latter specifies the number of sample points used to interpolate individual fuzzy sets and determines the precision of fuzzy logic operations and the defuzzification process.

Figure 3 shows the fuzzy logic evaluation cost for a rule base with nine rules, five inputs and four outputs when evaluated on a Sun Sparc Ultra-1.[2] To determine performance scaling as a function of the number of rules, we then replicated the entire rule base two and three times. The figure clearly shows that rule evaluation overhead is linear in both the number of rules and the fuzzy set resolution.

We believe most application and resource policy steering decisions will be realized on a time scale measured in seconds or minutes. Hence, based on the results of these preliminary experiments, the *Autopilot* prototype and fuzzy logic provide an effective infrastructure for closed loop adaptive control.

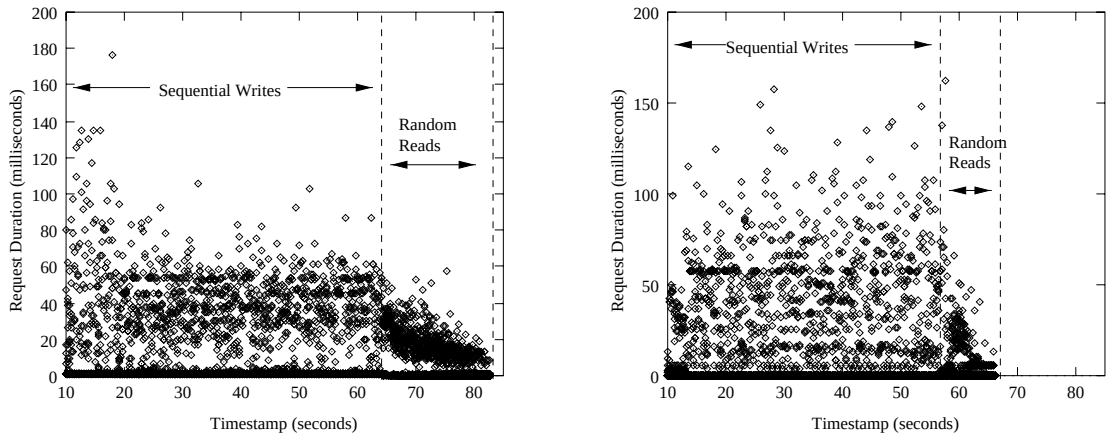
5. PPFS II: An Autopilot Testbed

Our recent characterization studies of parallel input/output patterns [1, 13, 12] have shown that parallel applications exhibit a wide variety of input/output request patterns, with both very small and very large request sizes, sequential and non-sequential access, and a variety of temporal variations. Because the interactions between these applications and the file system software change during and across application executions, it is difficult or impossible to determine a globally optimal input/output configuration or to statically configure runtime systems and resource management policies for parallel input/output.

Small input/output requests are best managed by aggregation, prefetching, caching, and write-behind,

[1] Indeed, *Autopilot* performance studies have led to more efficient Nexus implementations.

[2] This rule base defines a set of adaptive file caching policies used in the PPFS II parallel file system prototype described in §5.



(a) Non-adaptive (b) Autopilot Adaptive

Figure 4. PPFS II Input/Output Benchmark

though large requests are better served by streaming data directly to or from storage devices and application buffers. Complementary performance measurements using experimental parallel file systems [5, 11] confirmed that exploiting both runtime knowledge of input/output access patterns and real-time performance data to control data placement, caching, and prefetching could dramatically increase achievable input/output performance.

Based on these experiences, we are designing and implementing a second generation parallel file system, called PPFS II, that supports real-time adaptive control of file system policies and policy parameters. PPFS II is designed to operate atop either parallel systems or PC/workstation clusters and provides a flexible testbed for high-performance input/output experiments. It includes automatic behavioral classification techniques to identify and group application resource request patterns and a flexible set of fuzzy logic rule bases that can intelligently select file system policies based on input/output resource demands and supplies.

As an example of the potential power of an adaptive parallel file system using the *Autopilot* infrastructure, consider the caching policies in parallel input/output systems. These policies should ensure that the data that will be reused in the near future is retained in the cache, and that caching is disabled when it will degrade input/output performance.

Such a policy selection method is implemented in PPFS II using a fuzzy logic rule base that contains rules similar to those in Figure 5. This rule base obtains its inputs using *Autopilot* sensors, and policy parameters are changed using actuators. In this way, the caching system will adapt to the changes in the access pattern.

Figure 4 shows the effect of adaptive caching policies on performance.[3] For both experiments shown in the figure, a 128 MB file is first written sequentially in 16 KB units. Following this, a 32 MB section of the file is read randomly four times using 16 KB access sizes.

In the first experiment, PPFS II uses default caching parameters: 4 KB cache blocks and a 16 MB client cache. In the second experiment, a fuzzy logic decision procedure continuously monitors the file system using *Autopilot* sensors and obtains access pattern information through a neural network classification. At the beginning of the first sequential pattern, PPFS II chooses a small cache with an MRU replacement policy for write-only sequential accesses. When the access pattern transitions to small, random reads, the decision procedure chooses a larger, 30 MB cache and 80 KB blocks.

A comparison of these figures reveals the advantages of choosing the cache parameters dynamically. Dynamic adaptation decreases the benchmark execution time from over 80 seconds to less than 70 seconds and also decreases the mean response time for each request.

6. Status and Future Work

Although the initial *Autopilot* prototype is operational, much work remains. We are developing a more extensive suite of sensors and sensor attached functions

[3] These experiments were conducted on a UltraSparc 1 with Solaris 2.5.1 and a Western Digital 4.3 GB SCSI-3 hard disk.

```
if ( ReadWriteMix  == READONLY &&
     Sequentiality == NONSEQUENTIAL &&
     RequestSize   == LARGE )
        { CachingEnable =  DISABLED; }

if ( ReadWriteMix  == READONLY &&
     Sequentiality == NONSEQUENTIAL &&
     RequestSize   == TINY )
        { CachingEnable =  ENABLED;
          CacheSize     =  HUGE;
          BlockSize     =  LARGE; }

      ...              ...

if ( ReadWriteMix  == WRITEONLY &&
     Sequentiality == SEQUENTIAL )
        { CachingEnable =  ENABLED;
          CacheSize     =  SMALL;
          BlockSize     =  LARGE;
          ReplacementPolicy = MOSTRECENTLYUSED; }
```

Figure 5. Adaptive I/O Rule Base

for real-time data reduction, coupling *Autopilot* with an immersive virtual environment [10] for interactive steering using the Pablo Self-Describing Data Format (SDDF) [9], and conducting an extensive set of performance studies using instrumented applications and libraries.

Of the later, the most ambitious is the application of *Autopilot* to the implementation of the PPFS II adaptive, parallel file system and experiments with interactive steering of distributed software via immersive virtual environments.

7. Acknowledgments

Ruth Aydt, Christopher Elford, Tara Madhyastha, and Eric Shaffer all contributed important ideas to the *Autopilot* design. We are also grateful to Ian Foster, Carl Kesselman, and Steve Tuecke for their guidance on the use of Nexus.

References

[1] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Characterization of a Suite of Input/Output Intensive Applications. In *Proceedings of Supercomputing '95*, Dec. 1995.

[2] I. Foster and C. Kesselman. *Computational Grids: The Future of High-Performance Distributed Computing.* Morgan-Kaufmann, 1998.

[3] I. Foster, C. Kesselman, and S. Tuecke. The Nexus Approach to Integrating Multithreading and Communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.

[4] W. Gu, J. Vetter, and K. Schwan. An Annotated Bibliography of Interactive Program Steering. *SIGPLAN Notices*, 29(9):140–8, 1994.

[5] J. V. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal. PPFS: A High-Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, July 1995.

[6] T. M. Madhyastha and D. A. Reed. Exploiting Global Input/Output Access Pattern Classification. In *Proceedings of Supercomputing '97*, Nov. 1997.

[7] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irwin, K. L. Karavanic, K. Kunchitkapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer*, 28(11):37–46, Nov. 1995.

[8] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS-2: The Second Generation of a Parallel Program Measurement System. *IEEE Tranactions on Computers*, 1(2):206–217, Apr. 1990.

[9] D. A. Reed. Experimental Performance Analysis of Parallel Systems: Techniques and Open Problems. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 25–51, May 1994.

[10] D. A. Reed, C. L. Elford, T. Madhyastha, W. H. Scullin, R. A. Aydt, and E. Smirni. I/O, Performance Analysis, and Performance Data Immersion. In *Proceedings of MASCOTS '96*, pages 1–12, Feb. 1996.

[11] D. A. Reed, C. L. Elford, T. Madhyastha, E. Smirni, and S. L. Lamm. The Next Frontier: Interactive and Closed Loop Performance Steering. In *Proceedings of the 1996 International Conference on Parallel Processing Workshop*, pages 20–31, August 1996.

[12] H. Simitci and D. A. Reed. A Comparison of Logical and Physical Parallel I/Oa Patterns. In *International Journal of Supercomputer Applications, to appear* 1998.

[13] E. Smirni, C. L. Elford, and D. A. Reed. Performance Modeling of a Parallel I/O System: An Application Driven Approach. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Mar. 1997.

[14] J. Vetter and K. Schwan. High Performance Computational Steering of Physical Simulations. In *Proc. Int'l Parallel Processing Symp.*, pages 128–132, Geneva, 1997.

[15] J. C. Yan. Performance Tuning with AIMS – An Automated Instrumentation and Monitoring System for Multicomputers. In *Proceeedings of the 27th Hawaii International Conference on System Sciences*, pages 625–633, Jan. 1994.

[16] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8(3):338–353, June 1965.