The Autopilot Performance-Directed Adaptive Control System

Randy L. Ribler, Huseyn Simitci, Daniel A. Reed

Department of Computer Science, University of Illinois, Urbana, Illinois 61801, USA

Abstract

High-performance computing is rapidly expanding to include distributed collections of heterogeneous sequential and parallel systems and irregular applications with complex, data dependent execution behavior and time varying resource demands. To provide adaptive resource management for dynamic applications, we are developing the Autopilot toolkit. Autopilot provides a flexible set of performance sensors, decision procedures, and policy actuators to realize adaptive control of applications and resource management policies on both parallel and wide area distributed systems.

Keywords: Performance Analysis. Adaptive Control. Application Steering.

1 Introduction

The scope of high-performance computing is rapidly expanding from single parallel systems to distributed collections of heterogeneous sequential and parallel systems. Moreover, emerging applications are irregular, with complex, data dependent execution behavior, and dynamic, with time varying resource demands. In consequence, application developers increasingly complain that even small changes in application structure can lead to large changes in observed performance.

1This work was supported in part by the Defense Advanced Research Projects Agency under DARPA contracts DABT63-94-C-0049 (SIO Initiative), DAVT63-91-C-0029 and DABT63-93-C-0040, F30602-96-C-0161, and DABT63-96-C-0027 by the National Science Foundation under grants NSF ASC 92-12369, 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.
The performance sensitivity of current parallel and distributed systems is a direct consequence of resource interaction complexity and the failure to recognize that resource allocation and management must evolve with applications, becoming more flexible and resilient to changing resource availability and resource demands. Currently, software developers are forced to engage in a time consuming cycle of program development, performance measurement, and tuning to create non-portable code that conforms to parallel and distributed system idiosyncrasies.

By integrating dynamic performance instrumentation and on-the-fly performance data reduction with configurable, malleable resource management algorithms and a real-time adaptive control mechanism, flexible runtime systems could automatically choose and configure resource management algorithms based on application request patterns and observed system performance. Such an adaptive resource management infrastructure would increase portability by allowing application and runtime libraries to adapt to disparate hardware and software platforms and would increase achieved performance by choosing and configuring those resource management algorithms best matched to temporally varying application behavior.

This view of adaptive runtime systems is buttressed by recent experiences with flexible input/output policies [12,10,19] and by adaptive runtime systems for wide area computing [6]. In both cases, use of real-time performance data to adapt to changing resource demands and availability has yielded order of magnitude performance improvements.

Based on this thesis, this paper describes the design and prototype implementation of the Autopilot real-time adaptive control infrastructure. Autopilot provides a flexible set of performance sensors, decision procedures, and policy actuators to realize adaptive control of applications and resource management policies on both parallel and wide area distributed systems.

The remainder of this paper is organized as follows. First, in §2, we describe the design goals of an adaptive control system, identify the key Autopilot components, and highlight important implementation constraints. In §3-§4, we describe the Autopilot sensor and classification infrastructure. This is followed in §5-§6 by a description of the actuator infrastructure and remote access to sensors and actuators. In §7, we describe flexible decision procedure mechanisms based on fuzzy logic, followed in §8 by a unified example of Autopilot's application to adaptive input/output systems. Finally, §9 summarizes the current state of Autopilot's development and plans for future work.
2 Adaptive Control System Design

During execution, parallel and distributed applications issue requests to the underlying runtime system, which responds by managing logical and physical system resources. As a quantitative basis for adaptive performance optimization, instrumentation sensors then capture salient aspects of both the application stimuli and system responses.

As a complement to quantitative performance data, knowledge of qualitative application behavior (e.g., sequential file access or latency-dominated communication), obtained from either user-written assertions or automatic classification techniques, is useful when dynamically choosing resource management policies. Quantitative data can then be used to dynamically adjust the parameters of the selected resource management policy.

Given quantitative and qualitative performance data, some resource decisions are best made locally because the requisite performance data was captured locally and the time constants for policy parameter adjustment may preclude remote decision making. Other decisions are best made globally, using knowledge of aggregate resource use and the behavior of all of an application’s tasks. Hence, a decision infrastructure must include both local and global components, with efficient mechanisms for routing subsets of the captured performance data from local performance sensors to one or more centralized decision sites.

Maximizing the performance of distributed and parallel computations often requires satisfying conflicting goals (e.g., minimizing latency and maximizing bandwidth). Hence, decision procedures must be sufficiently robust to choose policies based on potentially conflicting, approximate, and noisy data. Finally, to maximize portability, decision procedure infrastructure should be separated from specific policies and from a particular hardware environment. With appropriate decomposition, one can tune policies for a new system or behavioral regime simply by changing the decision procedures.

These guidelines for real-time measurement and adaptive control are based on our experience with the Pablo performance analysis environment [17,16] and extensions to support real-time performance monitoring, qualitative classification of file access patterns [13], and table-driven selection of file policies [19,12]. Building on these lessons and components of our existing Pablo software, we have designed and are implementing the Autopilot toolkit for real-time adaptive control of distributed and parallel computations. Below, we describe the components of the Autopilot toolkit and our implementation approach.
2.1 Autopilot Components

Figure 1 shows the basic infrastructure of the Autopilot toolkit. The individual components for distributed software control include:

- low overhead, distributed software sensors for real-time performance data capture and synthesis of quantitative and qualitative application and system performance metrics,
- interfaces for user-written assertions that can provide qualitative classifications of resource request patterns,
- automatic behavioral classification techniques to identify and group application resource request patterns both locally and globally,
- resource actuators that can enable resource allocation policies and configure policy parameters,
- distributed sensor/actuator managers for remote acquisition of sensor/actuator identifiers, enabling management of wide area computations,
- a set of fuzzy logic decision mechanisms that exploit real-time sensor inputs and behavioral classifications to dynamically select resource management policies, and
- flexible resource management policies that can respond to actuator inputs and adapt their behavior based on changing resource demands and resource availability.

Although designed to interoperate, the Autopilot components are implemented independently and can be used separately. Thus, one can use the sensor infrastructure to capture real-time performance data streams and drive displays of system activity without use of either the decision procedure or actuator components.
2.2 Implementation Issues

Given the basic design of §2.1, a host of implementation decisions remain. Of these, the most important are support for both shared and distributed memory programming models and the seamless integration of both local (single machine) and wide area (network) parallelism.

To maximize portability, Autopilot is built atop the successful Nexus [5] communication library and its unified model of a shared address space across local and wide area networks. Nexus includes portable, lightweight, communication mechanisms that support intraprocess, interprocess, and internode data sharing on sequential and parallel systems (both shared and distributed memory) and on distributed networks of systems via a global name space. Moreover, Nexus defines a platform-independent thread interface that hides the idiosyncrasies of specific thread packages.

In addition to being portable, sensors and actuators must operate with both source and object code. This implementation constraint is a consequence of the need to monitor both system and application performance metrics and the increasing use of proprietary libraries. Not only is proprietary system or library source code rarely accessible, the cost and overhead to build instrumented versions of operating systems or libraries is prohibitive. Moreover, the desired instrumentation points may not be known until execution time. Thus, it should be possible to patch object code, inserting calls to sensors and actuators as needed (e.g., using the Paradyn toolkit [14]).

To support a wide range of instrumentation requirements, sensors and actuators can operate in both threaded and non-threaded modes. In the threaded mode, a separate monitoring thread executes in the same address space as the entity being monitored, passively acquiring data by observing shared variables or changing values based on actuator commands. In the unthreaded mode, a sensor or actuator is invoked by a procedure call from the code being monitored.

Finally, sensors and actuators must be remotely accessible and controllable. Wide area computations execute on potentially heterogeneous collections of parallel systems that may be geographically dispersed. Monitoring and control software at remote sites must be able to acquire the identity of remote sensors and actuators, change their configurations, and retrieve data. In §3–§5, we will examine all of these issues in greater detail.
3 Flexible Performance Sensors

Historically, performance instrumentation required users, parsers, or compilers to insert instrumentation library calls in application source code. IPS-2 [15], AIMS [26]; Pablo [17] and other popular performance analysis toolkits all operated in this way. After instrumentation insertion, the application code was recompiled and performance data was collected during subsequent execution. While this approach has proven successful for off-line performance analysis, it has several liabilities, some of which are particularly debilitating when the goal is real-time adaptive control rather than a posteriori performance analysis.

Because the key performance metrics are often unknown prior to program execution, the temptation is to excessively instrument the code to minimize the number of executions needed to identify performance problems. Such indiscriminate source code instrumentation can generate large volumes of performance data, significantly perturb program behavior, and lead to incorrect assumptions about the performance of the uninstrumented program. Instead, performance sensors should be lightweight, activated only when needed, and acquire the minimum amount of needed data. Moreover, whenever possible, they should process performance data at its point of capture, minimizing transmission costs.

3.1 Sensor Insertion and Data Reduction

To remediate the limitations of procedure-based source code instrumentation, Autopilot sensors can gather data using two different mechanisms. The first of these is the conventional method; a sensor records data in response to procedure calls that have been inserted in the instrumented program. These calls are executed by the application program thread or threads.

The second data gathering mechanism creates a separate thread that periodically awakens, reads one or more application variables in the address space, and then returns to sleep. Sensor controls allow external monitoring tasks to dynamically adjust the slumber interval and the sensor data buffering policy.

Threaded and non-threaded modes are appropriate in different circumstances. Threaded monitoring can acquire data asynchronously, with the rate that data is recorded and the associated effect on application perturbation both independent of program execution. Threaded monitoring is, however, limited in its ability to provide information about the context where program variables were modified. Conversely, procedurally invoked non-threaded monitoring is dependent on program control flow, making data collection rates dependent on program behavior, but supplying complete contextual information.
To enable local data reduction, all Autopilot sensors support attached functions. These functions are invoked each time a sensor receives data and act as data filters, transforming the original data to an alternate, often reduced, form. Data reduction at the sensor level can provide a wide range of functions, from computing simple statistics (e.g., counts, means, minima, maxima, and distributions) to the transformation of a set of sensor readings to qualitative behavioral classifications.

As an example of the later, an attached function could process a sequence of file read/write offsets captured by a file input/output sensor and produce a qualitative classification of the file access pattern (i.e., identifying the access stream as sequential, strided, or random). As we shall see in §4, such high-level classifications provide critical guidance when selecting resource policies. This ability to extend the basic functionality of performance sensors is a generalization of that provided by the Pablo toolkit’s extension interface [16].

For dynamic instrumentation of object code, Autopilot is designed to exploit the Paradyn Metric Description Language (MDL) [14]. Using MDL, future versions of Autopilot will be able to insert threaded and procedural sensors in object code regardless of source code availability.

In addition to object code patching for sensor insertion and removal, Autopilot supports low overhead controls for enabling and disabling existing sensors. A disabled sensor will still produce some program perturbation, typically a procedure call or a periodic thread activation, but much less than an enabled sensor.

3.2 Sensor Monitoring

In addition to flexible sensor insertion and removal, as well as threaded and procedural activation modes, Autopilot sensors can transmit data using a variety of policies. These policies include

- **Transmit on demand:** The sensor immediately transmits the current sensor value(s) in response to a remote request. No additional transmissions occur until receipt of another request.
- **Periodic update:** Sensor data is transmitted at regular time intervals. The recipient specifies the desired interval, and a separate sensor thread is responsible for initiating the periodic transmissions.
- **Event-driven update:** Sensor data is transmitted in response to instrumentation statements which have been inserted into the application code.
- **Conditional update:** Data is transmitted only under prespecified conditions. For example, when the data lies within a particular range or when the sensor values exceed some threshold.
The rate that a sensor records data is independent of its transmission policy. Each time a sensor records data, it logs the data in a fixed length buffer. Regardless of the transmission policy in use, buffered data is always transmitted when the buffer fills. Thus, a sensor with a unit size buffer will transmit data each time it is invoked. Larger buffers trade sensor transmission latency for reduced total communication cost.

5.3 Sensor Example

As an example of the flexibility of Autopilot performance sensors, Figure 2 shows the key components of both procedural and threaded sensor utilization in the analysis of file read operations. In the figure, two sensors are used. The first, the ReadClassificationSensor, runs as a separate thread, monitoring the offsets of file reads and using the attached function IClassifier to generate qualitative classifications [13] of the file access pattern, identifying it as sequential, strided, or random.

The procedural ReadSizeSensor sensor uses a sliding window average attached function to generate a sequence of mean request sizes. The ReadSizeSensor sensor records data via the RecordData function invoked each time the UNIX read routine is called.

4 Hints and Classifications

Although sensors provide the quantitative data needed to make resource policy management decisions, our experience with adaptive file system policy selection [10] has shown that qualitative data on current and future resource demands is an effective complement. This information can be obtained either via user-supplied hints (i.e., assertions about future resource use) or by synthesizing qualitative resource descriptions from quantitative data (e.g., as in the example of Figure 2).

Hints and automatic classification have complementary advantages and disadvantages. However, to maintain independence between sensors and decision procedures, both hints and classifications must describe resource use, but not specify desired actions. That is, hints and classifications must describe, not prescribe. Otherwise, portability is lost.
// Sensor constructors
IntegerSensorFunction ReadClassificationSensor( "ReadClassification",
    bufferSize, // Sensor local buffer size
    &IOClassifier, // Classification attached function
    ReadClassAttributes // Sensor attributes for identification
);

IntegerSensorFunction ReadSizeSensor("AverageReadSize",
    bufferSize, // Sensor local buffer size
    &SlidingWindowAverage, // Averaging attached function
    AverageReadsAttributes // Sensor attributes for identification
);

// Activate the threaded classification sensor
ReadClassification.StartThread();

// Instrumented file read routine
int read( int fileDescriptor, char *fileBuffer, int requestSize )
{
    :
    ReadSizeSensor.RecordData( requestSize );
    :
}

Fig. 2. Autopilot Sensor Example

4.1 User Resource Hints

The notion that applications might improve performance by providing advisory information to system software is not new. For example, UNIX systems have long allowed applications to issue hints about future memory reference patterns. Similarly, new application programming interfaces (APIs) for high-performance input/output include interfaces for user specification of future file access patterns.

For example, the Scalable I/O initiative’s low-level API [2] allows users to provide both ordered and unordered access hints. Ordered hints enumerate a list of future accesses and the order they will occur. Conversely, unordered hints qualitatively describe future access patterns without ordering specific requests (e.g., by specifying that file accesses will be random). Because such hints are qualitative, they are largely machine independent, making it possible to move codes across systems without assertion changes, as would be required if the assertions stated numerical conditions (e.g., input/output request rates).

From Autopilot’s perspective, hints are simply user-generated sensor values.
Via the sensor procedural interface, hints define data passed directly to regist-tered recipients.

4.2 Qualitative Classifications

For motivated and savvy software developers, powerful APIs and hint inter-
faces make possible significant performance optimizations. However, based on
our instrumentation of large research codes and subsequent discussions with
the code developers, we have found that developers are often either unaware of
their application dynamics or are surprised by the interaction of request pat-
terns with system resource management policies. Hence, we believe automatic
qualitative classification of application requests is needed to identify patterns
and choose policy families.

Such qualitative classifications process quantitative sensor data as input, gen-
erating high-level, qualitative descriptions as output. Because the classification
process is automatic, it lessens the application programming burden. More-
over, when users do supply hints, the classification process can verify hint
accuracy and educate users about application behavior by reporting actual
qualitative behavior.

To support automatic, qualitative classification of resource use, we have de-
veloped a suite of trained artificial neural networks (ANNs) and hidden Markov
models (HMMs) [13,11]. Within the Autopilot framework, our ANN/HMM
classification software operates as a group of sensor attached functions, inter-
cepting application input/output parameters, and reporting qualitative clas-
sifications to sensor clients.

Based on our experience classifying input/output patterns [13,11], we believe
that both ANNs and HMMs are necessary — each is suited to a different type
of classification problem. After offline training, ANNs can efficiently classify
access streams in real-time (e.g., identifying input/output requests as strided
and read-only). However, they cannot identify new patterns, merely estimate
the proximity of the request stream to previously defined patterns.

In contrast, HMMs build a probabilistic model of the access stream, predict-
ing the likelihood of future accesses. This generality allows HMMs to classify
arbitrary access patterns, albeit at the expense of a prior execution for train-
ing. Subsequent executions can then use the probabilistic model for resource
prediction.
// Actuator constructor for prefetch control
IntegerActuator enablePrefetch("enablePrefetch",
    &prefetchEnabled,  // Local (boolean) file system variable
    enablePrefetchAttributes  // Actuator attributes for identification
);

// Actuator constructor for prefetch data volume
IntegerActuator setPrefetch("setPrefetch",
    &prefetchCount,   // File prefetch block count
    setPrefetchAttributes,  // Actuator attributes for identification
    computePrefetchCount   // Attached function to compute prefetch
);

while (...) {
    if (prefetchEnabled) {
        setPrefetch.DisableActuatorUpdates();  // Set lock

        bytesRead = prefetchCount * blockSize;

        // If prefetchCount is changed here, totalBlocksPrefetched
        // would be calculated incorrectly.

        totalBlocksPrefetched += prefetchCount;

        setPrefetch.EnableActuatorUpdates();  // Release lock
    }
}

Fig. 3. Autopilot Actuator Example

5 Distributed Resource Actuators

The actual system modifications needed to produce desired system adaptations are achieved via use of actuators. Actuators are remotely controlled functions that can change local variable values and invoke local functions. Using actuators, a remote process can realize a wide variety of possible changes in the behavior of an instrumented application.

Figure 3 shows a small code fragment from an instrumented file system that can adaptively prefetch data based on signals from an external decision procedure (e.g., based on the classification and sensor data from Figure 2). An actuator named enablePrefetch is created to asynchronously manipulate
the value of the boolean variable `prefetchEnabled`. Another actuator named `setPrefetch` synchronously controls the value of the variable `prefetchCount`.

Notice that the value of boolean `prefetchEnabled` can be changed at any time via a simple write from external sources with no concern for data races. Conversely, the `prefetchCount` variable is used multiple times during each loop iteration. Fully asynchronous changes would lead to incorrect behavior. Hence, asynchronous updates are disabled for the critical section of the loop. Finally, determining the number of blocks to prefetch depends on local information that might not be available remotely. Hence, the `setPrefetch` actuator relies on an attached function that computes the actual prefetch amount.

6 Sensor/Actuator Managers

Based on Nexus global pointers, sensors and actuators provide the mechanism needed to monitor and control both local and remote tasks. However, acquiring data from the sensors in a remote task or issuing commands to that task's actuators requires one to know (a) the identity of the remote task(s), (b) the state and names of the active sensors and actuators, and (c) the actuator and sensor input and output data types.

When a system contains a large number of sensors and actuators, maintaining this metadata is a major software overhead. Similarly, when the sensors and actuators reside in software on a remote, network-connected system, even learning sensor/actuator identities may be problematic. To simplify metadata maintenance and remote monitoring, we developed a sensor/actuator manager and associated set of remote access functions for use by remote clients.

6.1 Remote Access Infrastructure

Figure 4 shows the interactions among the `Autopilot` sensor/actuator manager, a remote client task, and a remotely controlled task. The sensor/actuator manager is a repository of extant sensors and actuators and functions as a network accessible name server. In turn, `Autopilot` clients are tasks that receive services via sensors or actuators. Clients receive data from sensors in remote tasks, request changes in remote tasks by invoking actuators, and set parameters for both sensors and actuators.

To obtain a global pointer to a remote sensor or actuator, a client task contacts the manager via a standard WWW uniform resource locator (URL) and requests access to one or more sensors and actuators with specified attributes.
// Local client handler for remote sensor
void ReadClassificationHandler(nexus_endpoint_t endpoint,
    nexus_buffer_t dataBuffer,
    nexus_bool_t called_from_non_threaded_handler)
{
    // Call access pattern decision procedure with remote data
    updateIntAssertion("AccessClassification", dataBuffer);
}

main()
{
    IntegerSensorClient ReadClassification("ReadClassification",
        ReadClassificationHandler, // Callback for remote sensor
        ReadClassAttributes // Sensor attributes for selection
    );

    ReadClassification.StopThread(); // Stop handler thread
    ReadClassification.SetReadInterval( 2 ); // Change access interval
    ReadClassification.StartThread(); // Restart handler thread
}

Fig. 4. Autopilot Manager Interactions

Fig. 5. Autopilot Sensor Remote Control Example

The manager then returns a set of global pointers to the sensors and actuators and an associated set of metadata.

After retrieving a global pointer from the sensor manager, the client can then register with the remote sensor or actuator via an asynchronous Remote Service Routine (RSR) (i.e., a Nexus remote procedure call [5]). Following this, the client can communicate directly with the remote sensor or actuator.
6.2 Remote Control Example

Figure 5 shows the use of a sensor client that requests connection to all sensors whose attributes match those specified by the ReadClassAttributes variable (i.e., those of the sensor of Figure 2). The interaction and pattern matching with the sensor/actuator manager is encapsulated in the IntegerSensorClient interface. In the figure, the sensor client establishes direct communication with the sensor named ReadClassification (i.e., the sensor of Figure 2) and assigns the handler ReadClassificationsHandler as a callback function. This callback is invoked each time data is transmitted from the sensor. In the example code, the handler is used to update the value of an input to a decision procedure, described below.

The client can also remotely control the sensor's behavior. In this example, the calls to the SetReadInterval, and StartThread routines change the rate that the remote sensor transmits data. A complementary set of functions control data buffering and recording policies.

7 Fuzzy Decision Procedures

The Autopilot decision infrastructure accepts data from distributed sensors as inputs and relies on actuators to realize the results of the decision process. There are a variety of classic techniques for implementing such distributed decision mechanisms, ranging from decision tables and trees to standard control theory.

A standard implementation of decision tables for resource management would typically contain one dimension for each of the key performance sensor values (e.g., file read request sizes and cache hit ratios). In turn, each dimension would be partitioned based on the number of significant operating regimes (e.g., small, medium, and large read requests), and a policy and its associated parameters would be associated with each table entry. During use, policies are identified via table lookup using current sensor values.

Although such decision tables are sufficient to fully discretize the sensor space and associate policies with each point, not only do the storage requirements grow rapidly with the number of parameters, identifying the appropriate policies for each point is costly. 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 con-
sistent parameter space division, fuzzy logic targets precisely the attributes of the resource management problem that challenge classic techniques [28], namely conflicting goals and poorly understood optimization spaces. The overall system allows manipulation of linguistically described concepts through use of common sense knowledge [27] (e.g., file prefetching benefits small, sequential reads).

7.1 Fuzzy Logic Infrastructure

Figure 6 shows the basic flow of information through the Autopilot fuzzy logic decision mechanism. A fuzzy controller relies on fuzzy sets to represent the semantic properties of each input (sensor) and output (actuator). The variables are processed by a set of IF-THEN production rules that map the input values to the output space through a collection of fuzzy sets. These input sensor values may represent raw sensor metrics (i.e., unchanged since their point of capture), synthesized sensor metrics processed by one or more sensor attached functions, user hints, qualitative classifications, or any combination of these.

Unlike boolean logic values that are either true or false, the fuzzy variables of Figure 6 can assume any value ranging from 0 (false) to 1 (true) with a variety of different transition functions from 0 to 1. For example, UTILIZATION is a fuzzy variable with a smooth transition from HIGH (completely true) if devices are always busy to LOW (completely false) if devices are always idle.

Fuzzy logic theory supports a wide variety of transition functions for fuzzy variables and many functions for the fuzzy analog of boolean operators [8]. These fuzzy operators combine continuous fuzzy variables to yield a fuzzy output (e.g., a fuzzy AND might take the minimum of two fuzzy logic values across their ranges).

In consequence, multiple fuzzy rules can be partially true simultaneously. The antecedent of each IF-THEN rule is first computed, yielding a fuzzy logic truth value. Then, the truth of the conclusion is derived by scaling each rule's conclusion by the degree of truth of the antecedent. Finally, taking the union of the outputs of all rules with the same consequent yields the actual result.

The appeal of the fuzzy logic approach to policy control is that policy transitions are smooth, rather than discrete as with decision tables or trees. Hence, one can quickly and easily change the rule set or the functions describing truth values to rapidly explore a variety of different resource policies.
7.2 Decision Procedure Example

Figure 7 illustrates a simple set of fuzzy logic rules that might be used to control file prefetching in an adaptive input/output system. *Autopilot* sensors provide a time-varying stream of file read access classifications (e.g., via the sensors of Figure 2 and its remote interface in Figure 5). After fuzzification, this sensor data stream defines the value of the ReadClassification fuzzy variable.

In the figure, the rules whose conditions are non-zero (i.e., at least partially true) all contribute to determining the value of the output fuzzy variable PrefetchingFactor. After defuzzification, the value of PrefetchingFactor
if ReadClassification = SEQUENTIAL then
  PrefetchingFactor = HIGH

if ReadClassification = RANDOM then
  PrefetchingFactor = LOW

if ReadClassification = UNKNOWN then
  PrefetchingFactor = MEDIUM

Fig. 7. Fuzzy Logic File Prefetching Rules

defines the action taken by an Autopilot actuator to adjust the number of blocks that are prefetched (i.e., the actuators of Figure 3).

The advantage of decoupling decision mechanisms from sensors and actuators should now be clear. The rule set of Figure 7 is architecture independent; neither the source of fuzzy inputs nor the sink of the fuzzy outputs is specified. The value ReadClassification is an abstraction whose values can be bound to a sensor value, a user hint, or even the output of another decision procedure. Similarly, PrefetchingFactor is an abstraction of an actuator, with no implicit mapping.

The decision procedure inputs and outputs are bound to specific sensors and actuators via global pointers from the sensor/actuator manager. By changing the mapping, one can apply the rule set using different sensors, choose different policies, or even control different systems.

8 Parallel Input/Output Example

Our recent characterization studies of parallel input/output patterns [3,18,24,25,23,22] 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. Small input/output requests are best managed by aggregation, prefetching, caching, and write-behind, 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 [7,19] 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.
As an example of the potential power of an adaptive parallel file system using the *Autopilot* infrastructure, consider a parallel application on \( P \) processors that first writes an \( N \) block data file using a strided access pattern (i.e., each processor writes a distinct set of blocks with the same fixed stride). Later,\(^1\) each processor \( p \) repeatedly and sequentially reads the subset of file blocks in the range \([\frac{EN}{P}, \frac{(E+1)N}{P}]\).

In isolation, each processor's write pattern appears strided, yet when examined collectively, it is clear that the processors collectively write the file sequentially. A qualitative access pattern classifier \([12,10,13]\) could locally classify each access pattern as strided, then combine the strided classifications to identify the globally sequential pattern.

Because the access patterns may be temporally skewed across processors (e.g., due to load imbalances) or not all processors may be engaged in input/output, such a global classification is, by necessity, approximate. Thus, the decision mechanism must be sufficiently robust to choose file policies based on potentially conflicting or approximate information (i.e., the access pattern is "almost" sequential or only a subset of the processors read or write data). Fuzzy logic was created for precisely this purpose and can efficiently control systems even when sensor data is noisy or variable.

Using the global access pattern classification, a file policy selection and configuration mechanism based on fuzzy logic might select a write back policy that merges file writes from each processor, forming larger, contiguous blocks for sequential write back to secondary storage devices. Using performance data on disk queue lengths and response times, captured by real-time performance sensors, the fuzzy logic controls would use policy actuators to choose the write back policy and optimally choose the size of the write back units.

For large block writes of the form just described, standard disk striping is ideal — it exploits multiple disks to reduce write transfer time. However, for the application's subsequent file read phase, the striping pattern means that all processors will contend with all others on each read. Distributing the file blocks such that each processor accesses a set of independent disks would reduce the read access time. This approach is based on Chen and Patterson's \([1]\) observation that the notions of parallelism (i.e., the number of disks across which a file is striped) and concurrency (i.e., the number of outstanding input/output requests) are mutually conflicting.

With simple fuzzy rules of the form

\[
\text{if Concurrency = HIGH then}
\]

\(^1\)Lest this pattern seem artificial, it is a simplified form of input/output patterns commonly found in parallel codes.
one can explicitly state this inverse relation and adaptively determine an optimal number of disks to stripe data for a particular file access pattern. Figure 8 illustrates a representative set of fuzzy values for such a rule set.

In addition, given knowledge of the future file read pattern, an adaptive file system might redundantly store the file blocks, creating a second copy of the file in a format efficient for parallel reads. This redundant copy might be created concurrently with the original (e.g., if the writes are sufficiently bursty to allow file data rearrangement during idle periods) or later, during an idle period. Such a redundant storage scheme would trade disk capacity for bandwidth by allowing the file system to access either or both copies during reads.

Such adaptive file placement decisions become extremely important on large scale parallel systems (e.g., planned systems in the U.S. Department of Energy Accelerated Strategic Computing Initiative (ASCI) [4] with over 10,000 disks and a few thousand processors). On such systems, it is highly likely that application file access patterns (request sizes and access concurrency) will dictate adaptive distribution of files across subsets of the storage devices.

9 Current Status and Plans

At this writing, we have completed implementation of prototype sensor, actuator and remote client libraries, and we are developing a fuzzy-logic inference engine to support decision procedures. During the coming months, we will be extending the basic prototype to include a richer set of sensors and actuators, completing support for object code patching, and integrating selected components from the Pablo performance analysis toolkit [17,16].
As an *Autopilot* validation test, we are developing an adaptive input/output library, based on the Scalable I/O Initiative's low-level API, that uses *Autopilot* sensors to capture input/output patterns, performance decision procedures to select and control caching, prefetching, and data placement policies, and actuators to implement policy decisions. This adaptive input/output system, called PPFS II, will be completed in two phases. The first phase, now underway, focuses on single processor input/output. The second phase will implement a parallel adaptive file system that embodies the features described in §8.

The *Autopilot* sensor/actuator/decision procedure model makes possible closed loop, adaptive control of distributed and parallel applications and runtime systems. By replacing decision procedures with real-time visualizations, *Autopilot* sensors and actuators can support interactive performance steering. In a complementary project, we are developing a collaborative virtual environment for direct manipulation of software structure and dynamics. Building on our Avatar virtual environment [20,21,9] experiences, this new environment, called *Virtue*, displays real-time data streams from *Autopilot* sensors, allows users to interactively choose resource policies and adjust policy parameters, and instantiates virtual environment manipulations via *Autopilot* actuators.

Acknowledgements

Many of the *Autopilot* ideas and functionality are the result of fruitful discussions with Ruth Aydt, Christopher Elford, Tara Madhyastha, James Oly, and Michael Mesnier.

References


