January 1998

SWiFT : a feedback control and dynamic reconfiguration toolkit

Ashvin Goel

David Steere

Calton Pu

Jonathan Walpole

SWiFT: A Feedback Control and Dynamic Reconfiguration Toolkit

Ashvin Goel, David Steere, Calton Pu, Jonathan Walpole
Department of Computer Science and Engineering
Oregon Graduate Institute, Portland

Abstract

We introduce SWiFT, a toolkit for building adaptive system software using a control-theoretic approach. SWiFT allows systematic implementation of feedback-control mechanisms. It also provides a framework for composing simple feedback mechanisms that operate within limited domains, and for dynamically reconfiguring them. This composition allows the application to adapt efficiently across a wide range of operating conditions. We describe a streaming application to demonstrate the feasibility of this technology.

1 Introduction

In this paper we advocate a systematic approach for building adaptive system software based on feedback-control theory, and present a toolkit that incorporates this approach. Feedback control helps produce predictable control and monitoring components. It requires the control goal and design specifications to be clearly stated, thus allowing analysis of properties such as stability. Our approach allows us to leverage the existing body of knowledge in hardware control for controlling software systems.

Our goal is to move the task of building adaptive system software from wizardry to engineering. Currently, feedback controls of software systems are brittle and written in an adhoc manner. As a result, it is difficult to move an existing control, such as TCP flow control [11], to a new domain such as CPU scheduling. In addition, existing controls are built with implicit assumptions about the system's run-time environment and can become unstable in the face of large or discontinuous variations in the operating environment.

SWiFT addresses these problems by providing a framework and methodology for building controls that are modular, dynamically reconfigurable, and predictable. Modularity results from our use of components and containers as the underlying abstraction. SWiFT enables dynamic reconfiguration by limiting the interaction between components to a simple input/output model and by supporting guarding and replugging of controllers [14]. SWiFT supports predictability by providing analysis tools based on control theory, and a domain-specific language for specifying composition with predictable results. In addition, SWiFT provides GUI-based debugging tools such as a software oscilloscope and a library of feedback components such as low pass filters to ease the task of building adaptive system software.

This project was supported in part by DARPA contracts/grants N66001-97-C-8522, N66001-97-C-8523, and F19628-95-C-0193, and by Tektronix, Inc. and Intel Corporation.

We have implemented SWiFT in C++ and Java and we have applied it to user-level applications running on Windows NT. Version 1.0 of SWiFT is available (along with a tutorial) at http://www.cse.ogi.edu/DISC/projects/swift. We are currently developing adaptive control mechanisms in a diverse range of domains on NT, from network flow control in multimedia streams to proportion-based CPU scheduling. These applications of SWiFT are discussed in more detail in Section 5. In addition, we are building a visual editor for designing, implementing, and monitoring controls using SWiFT.

The next section describes related work. Then Section 3 presents the feedback-control model in SWiFT. Section 4 provides an overview of run-time reconfiguration of control components and Section 4.1 describes an application of reconfiguration. Finally in Section 5, we describe our future plans for SWiFT, and the types of applications where we intend to apply SWiFT.

2 Related Work

The ideas in SWiFT are indebted to previous work on feedback-based control systems. Massalin and Pu introduced the idea of feedback-based resource management in operating systems in the Synthesis kernel [15]. Cen built an early version of SWiFT, and used it to build an adaptive distributed multimedia player [4]. We have extended Cen's toolkit, ported it to NT, provided composition and analysis tools, and incorporated a run-time reconfiguration mechanism. These extensions are discussed later in this paper.

Several commercially available toolkits, such as Matlab [18] and MATRIX$_X$ [9] support building linear [1], nonlinear [7] and fuzzy [13] controllers. They provide various predefined control building blocks, simulation, analysis and GUI tools. The target applications of these toolkits are traditional hardware or embedded control systems that have predictable dynamics and gradual transitions. These toolkits are designed to be used off-line at control design time, whereas SWiFT is designed for on-line run-time use. For example, SWiFT supports dynamic reconfiguration through guarding and replugging, and direct manipulation of a running control through its debugging tools.

Software feedback has been used extensively for adaptive scheduling, flow and congestion control [11, 17, 12, 2] and intra- and inter-stream synchronization in distributed multimedia systems [16, 5]. SWiFT allows many of these mech-

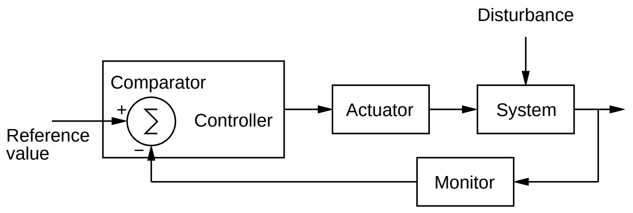

Figure 1. A block diagram of feedback control

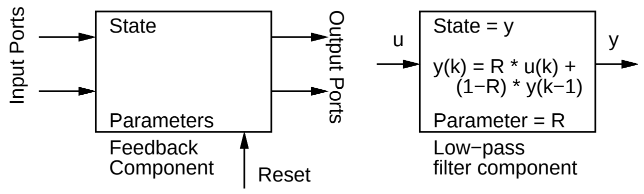

Figure 2. The SWiFT component model, and an example low-pass filter component.

anisms to be built systematically, simulated and visualized.

The mechanisms used for dynamic reconfiguration in SWiFT are similar to code replacement in specialization [19, 14] and to object invocation in multi-dispatch object-oriented languages [6], but the goals are different. Code specialization improves performance by replugging code that is partially evaluated given invariants in the current environment, whereas SWiFT reconfigures policies tuned for the current environment. Multi-dispatch languages can implement reconfiguration of arbitrary code, but this generality disallows simultaneous composition of *multiple*, *distinct* functionality. Section 4.1 shows that SWiFT's simple control model allows simultaneous composition of multiple feedback mechanisms.

3 Feedback Control Using SWiFT

Figure 1 shows the abstract architecture of a feedback control system built with SWiFT. The *controller* helps the system maintain a reference value of a controlled variable, while reducing the system's sensitivity to disturbance. The control is integrated with the system through *monitors* and *actuators.* A monitor measures the controlled variable, and is the source of the feedback. The controller's output causes the actuator to adapt the system's behavior in response to disturbances, or changes in the system's environment. For example, a feedback-based flow controller may monitor network bandwidth and latency, and drive an actuator that adjusts the congestion window size.

The design of a feedback control system separates the system from the control, the monitor, and the actuator, thus providing a modular design. Given a controller, one can determine a feedback control equation that specifies the controller's characteristic behavior. A controller may consist of a combination of feedback controls. Using standard control theory, SWiFT can determine the behavior of the composition of these simple controls by combining the equations for the simple controls. Hence, our approach results in predictable, composable, and modular control designs.

As an example of using this approach, consider the task of designing a network flow controller. One starts with a parameterized model of the system's environment, and then designs a control policy that tunes the system's behavior to the model. In this example, one can model the client's received packet rate, *C*, as varying linearly with the server's send rate, *S*, in the network. If *S* is less than the network's available bandwidth *B*, no packets are dropped and *C* equals *S*. When *S* exceeds *B*, packets are lost due to congestion and *C* is less than *S* (and probably less than *B* as well). In this example, the job of the controller is to tune *S* to approximate *B* by monitoring the rate of packet loss. Given the equation describing the relationship between *C*, *S*, and *B*, we can design an appropriate controller.

3.1 SWiFT's Abstractions

The basic blocks in SWiFT are *feedback components*. Feedback components read data from their *input port*(s), calculate an output value based on their characteristic behavior, and pass the value to their *output port*. A control circuit is built by connecting a component's output port to input ports of one or more components. Each output port can be connected to one or more input ports, but each input port can be connected to at most one output port. Monitors and actuators are special feedback components with no input ports and no output ports respectively.

The characteristic behavior of a feedback component can be adjusted by modifying the component's *parameters*. Parameters are constants in a component's characteristic equation, so changing a parameter requires recalculating the effect of this component on the controller's behavior. To avoid frequent recalculation, parameters are typically controlled from outside the controller, such as through a slider in the GUI. The *state* of a component is internal and generally not exposed by the component. A *reset* port is provided to reinitialize the component's state.

Figure 2 shows the feedback component model and a first-order low-pass filter component as an example. The output of the low-pass filter is an estimator of the average of its recent inputs. The parameter *R* is an aging factor that determines the contribution of old inputs to the average. The internal state is the previous output of the filter.

Feedback containers, shown in Figure 3, provide modularity and hierarchical structure. A container is a feedback component that contains other feedback components and containers, and defines a circuit of connections among its children and its input and output ports. The container's control equation is calculated from its circuit and sub-components (or children) using standard control theory. The container's parameters can be directly mapped onto the pa-

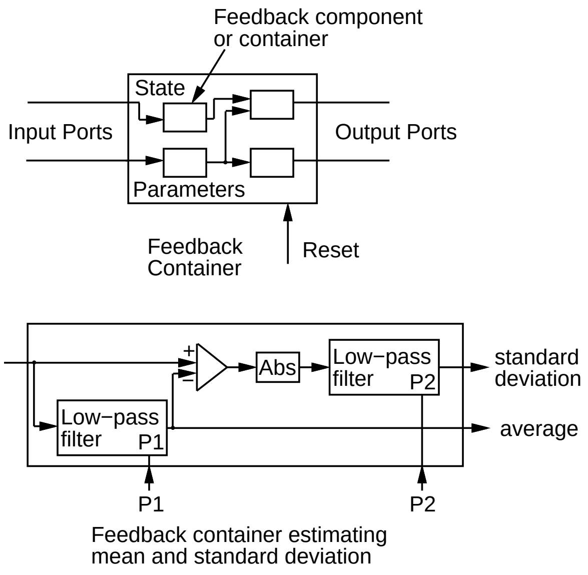

Figure 3. The container model, and an example container that estimates the average and the standard deviation.

rameters of its children.

Figure 3 shows an example of a feedback container that calculates the mean and the standard deviation of an input signal. The parameters of the low-pass filters are exposed by the container.

We distinguish the outermost, or top-level, container and allow it to manipulate and drive the lower layers. The top-level container may have its own thread (if it is asynchronous to the system it controls), and may or may not contain the monitors and actuators used to interact with the system. All its children must run synchronously with it, to avoid non-deterministic controller behavior. Hence a monitor or actuator that needs to run asynchronously from the controller, e.g., to run at a different rate, should be implemented as a separate top-level container connected to the controllers input or output port. These top-level containers may need to interact with code that is outside the SWiFT model. Therefore they are designed to interact with multiple component models such as Java Beans and COM. This generality is not needed in the inner components and containers, and hence is not provided.

3.2 Analysis and Debugging Tools

SWiFT currently performs simple analysis for feedback controllers. A component's transfer function (or characteristic equation) is specified by its creator. A container's transfer function is calculated from its internal layout and the transfer functions of its children. We use MuPad, a symbolic manipulation software, for doing the associated algebra.

Along with feedback analysis, SWiFT also helps visualize the outputs of a feedback controller in real time with an oscilloscope shown in Figure 4. We use asynchronous communication between the feedback controller and the GUI so

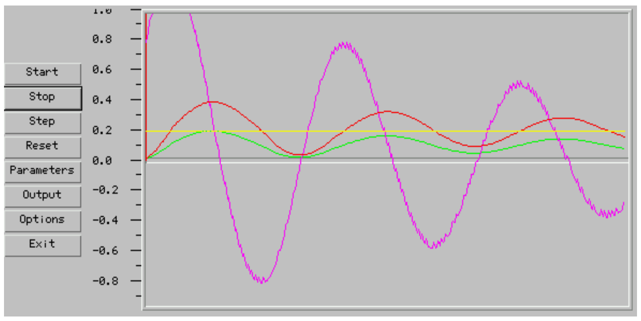

Figure 4. A snapshot of the SWiFT scope.

that the GUI does not become a bottleneck. Other GUI components in SWiFT include a control panel for accessing the parameters of the controller, a scope panel that allows adjustment of the outputs as shown on the scope and various signal generators such as sinusoid, square and random wave generators.

Currently, a C++ and a Java version of SWiFT exists for NT. There are also two sets of GUI components. We initially decided to implement both versions because we have been applying SWiFT to both new and legacy C and Java applications. Our eventual goal is to provide a high-level language for specifying controllers. These specifications will then be the basis for our analysis and composition tools, and could be used to generate code at run-time.

3.3 Feedback Control for Software Environments

While controllers built using SWiFT components can be analyzed for their stability characteristics, a complete stability characterization of a system requires modeling the behavior of the system, the monitor, and the actuator. This can be hard for several reasons. Software systems often do not satisfy the assumptions of simple feedback control, such as linearity and time-invariance, over their entire operating environment. Moreover, software environments can change significantly and control systems often degrade poorly when such drastic changes occur. Finally, users often change the control goal when resource availability changes significantly. SWiFT complements feedback control with dynamic reconfiguration to address these issues.

4 Dynamic Reconfiguration in SWiFT

Dynamic reconfiguration consists of tuning a control circuit's parameters, or replacing some or all of a control circuit at run-time. This reconfiguration is the *controlled* system's response to drastic changes in the underlying environment that violate the controller's basic design assumptions. For instance, TCP's adaptive flow-control algorithm performs poorly over wireless links. Replacing this policy with one more suited to wireless use results in better performance [20]. Dynamic reconfiguration is similar to hardware hot-swapping with the addition that the swapping is done automatically.

Reconfiguration is useful when the designer cannot completely describe the system's environment. This may happen because of a wide system operating environment or due to large resource variations. An example of the first situation is the use of TCP over wired and wireless links. An example of the second situation occurs when the available bandwidth changes significantly due to external load. In either situation, reconfiguration allows composing simple feedback mechanisms that operate well within limited domains. The system designer builds multiple controllers for the system for each limited domain, and then SWiFT configures the appropriate control for the current environment. If no control is appropriate, SWiFT can raise an exception notifying the system administrator or user of the error.

SWiFT can dynamically reconfigure the feedback components in the feedback controller, and can potentially manipulate the monitor and actuators as well. The reconfiguration is based on user-specified predicates on system properties, called *guards*. Guards are generally separate from the controller and must be explicitly programmed by the controller's designer.

Three types of reconfiguration are possible. First, a component parameter can be altered. Second, a reset that reinitializes the states and parameters of a component can be issued. As an example, the state of a low pass filter that is estimating current latency should be discarded after a network interface switch. Finally, new components can be plugged in or old components can be unplugged.

4.1 An Adaptive Streaming Application

As an example, we present the design of a feedback-based reconfigurable controller for streaming multimedia data over both lightly- and heavily-buffered networks [4]. The goal of the controller is to maximize throughput while avoiding jitter caused either by packet loss or by variation in latency. The controller, situated at the client, monitors the network and adjusts the server's send rate to achieve this goal. Our application is geared to streaming multimedia, so lost packets are not retransmitted.

The controller has three different modes of operation: start-up, and transmission over lightly- and heavily-buffered connections. The controller selects the proper mode for the current environment by dynamically reconfiguring itself. The start-up policy is similar to that used by TCP, and is not discussed further. In a lightly-buffered connection, e.g., over Ethernet, congestion rapidly leads to packet loss from buffer overflow, and the controller monitor's packet loss to detect congestion. In a heavily-buffered connection, such as PPP over a modem, congestion first results in substantially increased transmission latencies due to long queues with eventual packet loss due to buffer overflow. Hence the controller monitor's latency instead of packet loss.

The feedback policy for lightly-buffered connections adjusts the server's send rate, *S*, based on feedback from the client about its receive rate, *C*. The send rate at time *k+1*, S_{k+1}, is the client's receive rate at time *k*, C_k, plus a parameter δ. Hence the feedback control equation is $S_{k+1}=C_k+\delta$. We call this a *loss-feedback* policy since it is applicable when packets are being lost due to congestion. The *latency-feedback* policy, the policy for heavily-buffered networks, is $S_{k+1}=C_k+p(F-L_k)/T$, where *F* is the target buffer-fill level in the network, L_k is the measured buffer-fill level, *1/T* is the rate at which the controller is running, and the constant *p*, $0<p<1$, is used to provide stability. The derivation of this policy is left to the reader.

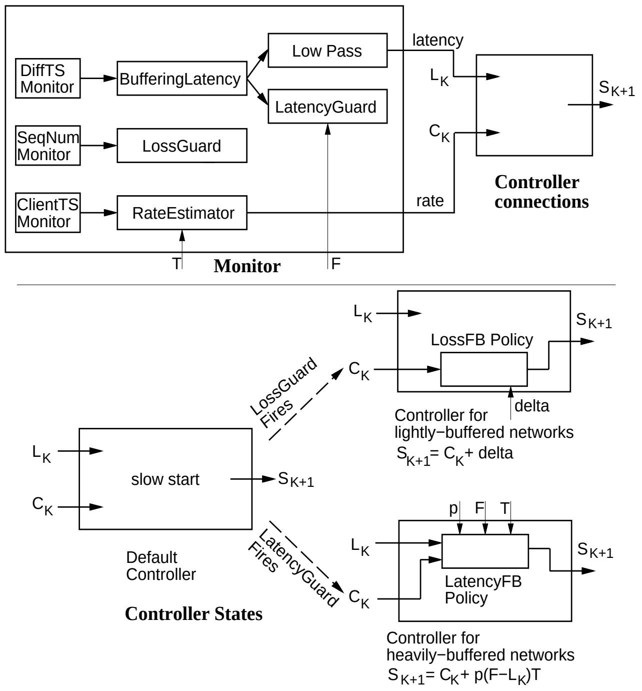

Figure 5. The monitor outputs an estimate of L_k, latency due to congestion, and C_k, the client's receive rate. Depending on the activation of the LossGuard or LatencyGuard, the controller calculates S_{k+1}, the desired server send rate, using the LossFB or LatencyFB policies respectively.

Figure 5 shows a block-diagram implementation of these policies in SWiFT. The monitor measures *DiffTS (*the end-to-end per-packet latency), *SeqNum (*the packet sequence number), and *ClientTS (*the packet's arrival time). It then calculates L_k and C_k, and passes them to the controller. The BufferingLatency container approximates L_k by taking the difference of current DiffTS with the minimum DiffTS (no buffering in the network) that it has seen, and the RateEstimator approximates C_k by averaging the inverse of the time between packet arrivals. The controller starts out with its slow-start policy. When the LatencyGuard detects increased latencies, it reconfigures itself to use the latency-feedback policy. If packets are lost, it again reconfigures to use the loss-feedback policy.

4.2 Reconfiguration Issues

Figure 6 shows a fragment of the SWiFT domain-specific program used to construct the monitor and the reconfigurable controller of Figure 5. Note the explicit predicates on the guards (in the definition of *RateMonitor*). The *disjoint* predicate indicates that the latency and the loss guards are not active at the same time. This implies that only three controller states are possible since the latency and the loss guards will not be active simultaneously. If the disjoint predicate is not stated, SWiFT will detect that the *LossFB* and the *LatencyFB* policies are both connected to the controller's output (S_{k+1} in Figure 5), and raise an error. One can enable concurrent use of both policies by adding a merger which passes the smaller of the two policy outputs to the controller's output.

Although SWiFT does *not* currently support run-time generation of controllers, the restricted interface of the feedback component model, and SWiFT's support for dynamic analysis of feedback containers, make this possible. Even without such a capability, the reconfiguration mechanism in SWiFT is useful because it provides explicit guards and a high-level mechanism for expressing multiple control mechanisms independently.

We omit a detailed description of the language shown in Figure 6 due to space constraints. However, we plan to include such a description in the full paper.

5 Using SWiFT on NT

We are currently working on extending SWiFT and using it to build adaptive resource managers on NT. SWiFT extensions include using run-time specialization to eliminate inefficiencies due to modularity, asynchronous guard invocation, real-time visualization of dynamic reconfiguration, more feedback analysis, and analysis for multiple, simultaneously operating feedback mechanisms for different system resources.

We are currently exploring the use of SWiFT on three diverse system control domains on NT: a streaming media player, an informed multimedia prefetching system, and a feedback-based proportional share CPU scheduler. The streaming media player delivers real-time multimedia content over unreserved bandwidth network links. Media filters, that can drop or duplicate packets, or perform increasingly lossy compression, can be inserted in the stream at the server or the client. The parameters of these filters can be changed in response to changing operating environments to utilize the available bandwidth effectively. For example, resolution can be changed on a frame-by-frame basis depending on the available bandwidth SWiFT helps build and analyze feedback mechanisms that adaptively adjust the parameters of these filters. We plan on basing this streaming player on Microsoft DirectShow's media infrastructure and using COM objects to interact with the feedback controllers in SWiFT.

```
RateMonitor extends FBMonitor {
  LatencyGuard latencyg;
  LossGuard lossg;
  RateMonitor(F, T) {
    (Inputs = 0, Outputs = 2);
    diffts = SystemMonitor(packet.DiffTS);
    seqnum = SystemMonitor(packet.SeqNum);
    clientts = SystemMonitor(packet.TS);
    bl = BufferingLatency();
    lpf = FOLowPassFilter();
    re = RateEstimator(T);
    latencyg = LatencyGuard(F);
    lossg = LossGuard();
    connections {
      bl(0) => lpf(0), latencyg(0);
      lfp(0) => Internal(0);
      re(0) => Internal(1);
      diffts(0) => bl(0);
      seqnum(0) => lossg(0);
      clientts(0) => re(0);
    }}
  GuardPredicates() {
    disjoint latencyg, lossg;
  }}

RateFeedback extends FBContainer {
  RateFeedback() {
    (Inputs = 2, Outputs = 1);
    -- slow start policy, details omitted here
  }}

RateFeedbackOnLoss reconfigures RateFeedback
when RateMonitor.lossg {
  RateFeedbackOnLoss(delta) {
    lofb = LossFB(delta);
    connections {
      lofb(0) => Internal(0);
      Internal(1) => lofb(0);
    }}}

RateFeedbackOnHighLatency reconfigures
RateFeedback
when RateMonitor.latencyg {
  RateFeedbackOnHighLatency(p, F, T) {
    lafb = LatencyFB(p, F, T);
    connections {
      Internal(0) => lafb(0);
      Internal(1) => lafb(1);
      lafb(0) => Internal(0);
    }}}
```

Figure 6. The definition of the monitor and the reconfigurable controller in SWiFT.

The prefetching system will use a feedback-based controller to control the amount of data prefetched from an application-defined view of a multimedia stream. This approach yields a well-behaved prefetching mechanism that is tuned to meet the dynamically changing needs of its application. For example, video scrubbing requires dynamically changing the stride in response to changes in play speed, e.g., slow forward vs. fast forward. A scrubber establishes a *view* of the video stream that corresponds to the current play speed, and the prefetcher uses its buffers to reduce application-visible latency and to respond quickly to changes in play speed. These goals requires a different policy mechanism, and hence uses SWiFT's dynamic reconfiguration. NT's modular design lets us implement this mechanism as a user-mode NT driver [8], allowing a single user-level controller to mediate the demands of multiple (possibly conflicting) applications.

The feedback-based proportional share CPU scheduler will make scheduling decisions for pipelines of processes that share data buffers. The controller monitors buffer fill levels and dynamically adjusts a process's share of the CPU and its period. The controller can exist as a user- or kernel-level driver, but the underlying scheduling mechanism needs to be integrated with the thread scheduler in the NT kernel. Although we could modify NT's abstraction layer as described by Carpenter et al. [3], an integrated approach would allow us to provide an adaptive universal thread scheduler, which is our ultimate goal. Unfortunately, we do not have access to the NT sources, and so have prototyped the basic mechanism in the Linux kernel, and will implement it in NT when we get access to the NT sources.

Although we could pursue this research on any platform, we chose NT for its modular structure, extensibility, and widespread use as a platform for multimedia applications. Our designs make heavy use of NT's user-level device drivers, Microsoft's DirectShow infrastructure, and may leverage Intel's Media Framework [10].

6 Conclusions

We have presented SWiFT, a software feedback toolkit that allows hierarchical composition of complex feedback systems based on simple building blocks. We have also introduced a means of dynamically reconfiguring feedback controllers and monitors. We have demonstrated the use of SWiFT to build an adaptive streaming application, and have discussed our plans for using SWiFT to build adaptive system resource managers on NT.

References

[1] William L. Brogan. *Modern Control Theory*. Quantum Publishers, Inc., 1974.

[2] Ingo Busse, Bernd Deffner, and Henning Schulzrinne. Dynamic QoS control of multimedia applications based on RTP. *Computer Communications*, 19:49–58, January 1996.

[3] Bill Carpenter, Mark Roman, Nick Vasilatos, and Myron Zimmerman. The RTX real-time subsystem for Windows NT. In USENIX, editor, *The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, pages 33–37, Berkeley, CA, USA, August 1997. USENIX.

[4] Shanwei Cen. *A Software Feedback Toolkit and Its Applications in Adaptive Multimedia Systems*. PhD thesis, Oregon Graduate Institute of Science and Technology, August 1997. Department of Computer Science and Technology.

[5] Shanwei Cen, Calton Pu, Richard Staehli, Cowan Cowan, and Jonathan Walpole. Demonstrating the effect of software feedback on a distributed real-time MPEG video audio player. In *Proceedings of the Third ACM International Multimedia Conference and Exhibition*, pages 239–240, San Francisco, CA, USA, November 1995.

[6] Craig Chambers. Predicate Classes. In O. Nierstrasz, editor, *Proceedings of the ECOOP'93 European Conference on Object-oriented Programming*, LNCS 707, pages 268–296, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[7] P. A. Cook. *Nonlinear Dynamical Systems*. Prentice-Hall International (UK) Ltd., 1986.

[8] Galen C. Hunt. Creating user-mode device drivers with a proxy. In USENIX, editor, *The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, pages 55–59, Berkeley, CA, USA, August 1997. USENIX.

[9] Integrated Systems, Inc. MATRIX$_X$ family technical specification. `http://www.isi.com/Products/MATRIXx/Techspec/toc.html`, 1997.

[10] Intel Corporation. Intel media framework, java edition. `http://developer.intel.com/ial/jmedia/JMFramework.htm`, 1997.

[11] V. Jacobson. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, 18, 4:314–329, 1988.

[12] Srinivasan Keshav. A control-theoretic approach to flow control. In *SIGCOMM'91*, pages 3–16, September 1991.

[13] F. Martin McNeill and Ellen Thro. *Fuzzy Logic: a Practical Approach*. Boston: AP Professional, 1994.

[14] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *SOSP'95*, pages 314–324, Copper Mountain resort, Colorado, USA, December 1995.

[15] Calton Pu, Henry Massalin, and John Loannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, Winter 1988.

[16] Srinivas Ramanathan and P. Venkat Rangan. Adaptive feedback techniques for synchronized multimedia retrieval over integrated networks. *IEEE/ACM Transactions on Networking*, 1(2):246–260, April 1993.

[17] Scott Shenker. A theoretical analysis of feedback flow control. In *Proceedings of SIGCOMM'90*, pages 156–165, September 1990.

[18] The MathWorks, Inc. Matlab product tour. `http://www.mathworks.com/products.html`, 1997.

[19] Eugen N. Volanschi, Charles Consel, and Crispin Cowan. Declarative specialization of object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, volume 32, 10 of *ACM SIGPLAN Notices*, pages 286–300, New York, October 1997. ACM Press.

[20] R. Yavatkar and N. Bhagwat. Improving end-to-end performance of TCP over mobile internetworks. In *Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, U.S., 1994.