Remote Application-level Processing through Derived Event Channels in ECho

Greg Eisenhauer
College of Computing
Georgia Institute of Technology

Abstract

Distributed and cooperative applications are becoming more common with the widespread adoption of network-centric computing models. One common design style for such systems is that of event generation and notification. This paper presents ECho, an event delivery middleware system that uses dynamic code generation to move application-level processing to remote locations in heterogeneous distributed systems. The resulting relocation of computation has potential to significantly reduce network and compute resource requirements for many applications.

1 Introduction

Distributed and cooperative applications are becoming more common with the widespread adoption of network-centric computing models. The implementation of complex systems in a network environment is often facilitated by adopting a loosely coupled design approach. One common design style for such systems is that of event generation and notification. Middleware infrastructure to support the event-based design style has been available for some time (*e.g.*, Yeast[10] and many others). Recent implementations and proposals have extended support to object-based systems (*e.g.*, CORBA Event Service[6] and and Java Distributed Events[8]. Event-based design is common in such important applications as collaborative visualizations, distributed virtual environments, distributed shared memory systems and quality of service infrastructures.

An important extension to the basic event model is support for event filtering or content-based subscription. This type of extension allows the receivers of events to specify the nature of the events they want to receive based on the content of data associated with the event. Ideally, such a filtering specification is applied at the event source where its use can dramatically reduce network traffic and bandwidth demands.

However, applying receiver-specified event filters at the event source is not necessarily a straightforward operation in a distributed heterogeneous event system. In this paper we describe event system middleware that uses dynamic code generation to efficiently implement application-level event filtering, processing and data reduction. Subsequent sections of this paper will describe the operation and interfaces of the base event system and the philosophy and implementation of the event processing extensions. Finally we present baseline performance numbers and discuss ongoing work.

2 The ECho Event System

2.1 Basic Functionality

ECho is an event delivery middleware system developed at Georgia Tech. Superficially, the semantics and organization of structures in ECho are similar to the Event Channels described by the CORBA Event Services

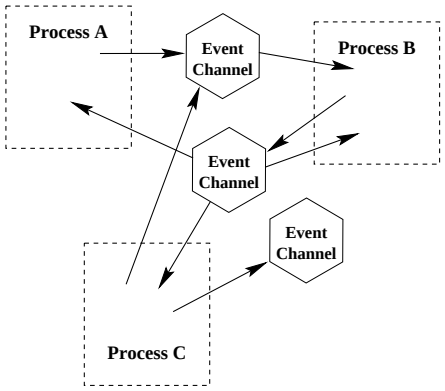

Figure 1: Processes using Event Channels for Communication.

specification[6]. Like most event systems, what ECho implements can be viewed as an anonymous group communication mechanism. In contrast with more familiar one-to-one send-receive communication models, data senders in anonymous group communication are unaware of the number or identity of data receivers. Instead, message sends are delivered to receivers according to the rules of the communication mechanism. In this case, event channels provide the mechanism for matching senders and receivers. Messages (or *events*) are sent via *sources* into *channels* which may have zero or more *subscribers* (or *sinks*). The locations of the sinks, which may be on the same machine or process as the sender, or anywhere else in the network, are immaterial to the sender. A program or system may create or use multiple event channels, and each subscriber receives only the messages sent to the channel to which it is subscribed. The network traffic for multiple channels is multiplexed over shared communications links, and channels themselves impose relatively low overhead. Instead of doing explicit *read()* operations, sink subscribers specify a subroutine (or handler) to be run whenever a message arrives. In this sense, event delivery is asynchronous and passive for the application.

Figure 1 depicts a set of processes communicating using event channels. The event channels are shown as existing in the space between processes, but in practice they are distributed entities, with bookkeeping data in each process where they are referenced. Channels are *created* once by some process, and *opened* anywhere else they are used. The process which creates the event channel is distinguished in that it is the contact point for other processes wishing to use the channel. The channel ID, which must be used to open the channel, contains the hostname and IP port number of the creating process (as well as information identifying the specific channel). However, event distribution is not centralized and there are no distinguished processes during event propagation. Event messages are always sent directly from an event source to all sinks.

Figure 2 summarizes the basic ECho API. ECho is implemented on top of DataExchange[3] and PBIO[2], packages developed at Georgia Tech to simplify connection management and heterogeneous binary data transfer. As such, it inherits from these packages easy portability to different network transport layers and threads packages. DataExchange and PBIO operate across the various versions of Unix and Windows NT, have been used over the TCP/IP, UDP, and ATM communication protocols and across both standard and specialized network links like ScramNet.

In addition to offering interprocess event delivery, ECho also offers mechanisms for associating threads with event handlers allowing a form of intra-process communication. Local and remote sinks may both appear on a channel, allowing inter- and intra-process communication to be freely mixed in a manner that is transparent to the event sender.

```
typedef struct _EChannel *EChannel;
typedef struct _ECSinkHandle *ECSinkHandle;
typedef struct _ECSourceHandle *ECSourceHandle;
typedef void (*ECHandlerFunction) (void *event, int length, void *client_data);

EChannel EChannel_create(DExchange de);
char *ECglobal_id(EChannel chan);
EChannel EChannel_open(DExchange de, char *global_id);

extern ECSourceHandle ECsource_subscribe ( EChannel chan );
extern ECSinkHandle ECsink_subscribe( EChannel chan, ECHandlerFunction func, void *client_data);
extern void ECsubmit_event ( ECSourceHandle handle, void * event, int event_length );
```

Figure 2: Basic event channel API.

2.2 Event Types and Typed Channels

One of the differentiating characteristics of ECho is its support for efficient transmission and handling of fully typed events. ECho allows types to be associated with event channels, sinks and sources. The set of supported types includes structures of atomic data types, NULL-terminated character strings and statically and dynamically sized arrays of these elements.[1] Knowledge of these event types allows ECho to perform the necessary marshalling and unmarshalling for safe binary data exchange in a heterogeneous environment.

A second distinguishing characteristic of ECho is that it supports the robust evolution of sets of programs communicating with events by allowing variation in data types associated with a single channel. In particular, ECho allows an event source to submit an event whose type is a superset of the event type associated with its channel. Conversely, an event sink may have a type that is a subset of the event type associated with its channel. This can be an extremely valuable feature when a system evolves because it means that event contents can be changed without the need to simultaneously upgrade every component to accommodate the new type. ECho even allows type variation in intraprocess communication, imposing no conversions when source and sink use identical types but performing the necessary transformations when source and sink types differ in content or layout. Support for typing on event channels, sources and sinks extends the ECho API with the routines depicted in Figure 3. The routines there use the type `IOFieldList`, a PBIO type that defines the layout of a structure by naming each field, along with the fields basic type, size and offset in the structure.[2] The `DEFormatList` type is simply a list of structure names and corresponding `IOFieldLists`. The types in this list specify any subtypes that might be used in the `IOFieldList`.

[1] These are the types supported by PBIO[2]. In the case of dynamically sized arrays, the array size is given by an integer-typed field in the record. In all cases the exact size, layout and extent of the data is known.

[2] Types associated with sinks and sources specify the actual layout of data on the architecture in use. Size and offset information for the types associated with a channel is ignored. Only the field names and types are used for superset/subset comparisons.

```
EChannel EChannel_typed_create(DExchange de, IOFieldList field_list, DEFormatList format_list);
extern ECSinkHandle ECsink_typed_subscribe(EChannel chan, IOFieldList field_list,
                                           DEFormatList format_list,
                                           ECTypedHandlerFunction func, void *client_data);
extern ECSourceHandle ECsource_typed_subscribe(EChannel chan, IOFieldList field_list,
                                               DEFormatList format_list);
extern void ECsubmit_typed_event(ECSourceHandle handle, void *event);
```

Figure 3: Typed event channel API.

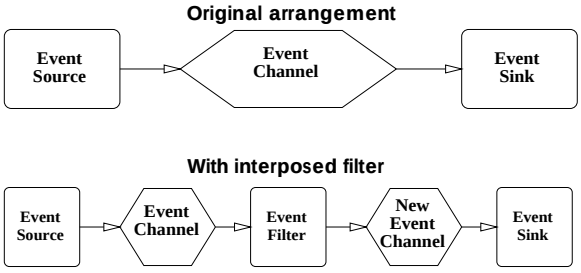

Figure 4: Source and sink with interposed event filter.

2.3 ECho Summary

ECho was initially developed as a data transport mechanism to support work in application-level program monitoring and steering of high-performance parallel and distributed applications[4]. In this environment, efficiency in transporting large amounts of data is of critical concern to avoid overly perturbing application execution. Because of this, ECho was designed to take careful advantage of DataExchange and PBIO features so that data copying is minimized. Even typed event transmissions require the creation of only small amounts of header data and require no copying of application data. Also, because ECho transmits events directly to sinks, it naturally suppresses event traffic when there are no listeners.

However, as in many other situations using event-based communication, program monitoring can produce large numbers of events which may overwhelm both the listeners and the intervening networks. This can be particularly frustrating if the listener is not interested in every byte of data that it receives. Unwanted events waste network resources in carrying the data, cause unnecessary perturbation to the application sending the data, and waste compute time for the listener who has to be interrupted, read, unpack and discard events he would rather not be bothered with. Using many event channels to subdivide dataflows is an effective and low-overhead of reducing unwanted traffic because listeners can limit their sink subscriptions to event channels carrying data that they want to receive. However, effective use of this technique requires the event sender have *a priori* knowledge of the appropriate subdivisions. The technique is also much more difficult to apply when a listener's definition of "unwanted event" depends upon the event content.

ECho's *Derived Event Channels* allow sink-specified event filtering, and even event data reduction, to be applied on the source end of event transmission. Performing these calculations at the source can be a win-win situation, reducing costs for both the sender and receiver and reducing bandwidth requirements on the intervening networks. The next section describes the Derived Event Channel abstraction and the critical role of dynamic code generation in performing these calculations in an efficient way in a heterogeneous environment.

3 Derived Event Channels

3.1 General Model

Consider the situation where an event channel sink is not really interested in *every* event submitted, but only wants every Nth event, or every event where a particular value in the data exceeds some threshold. Much compute and network overhead could be avoided if only the events of interest were transmitted. One way to approach the problem is to create a new event channel and interpose an event filter as shown in Figure 4.

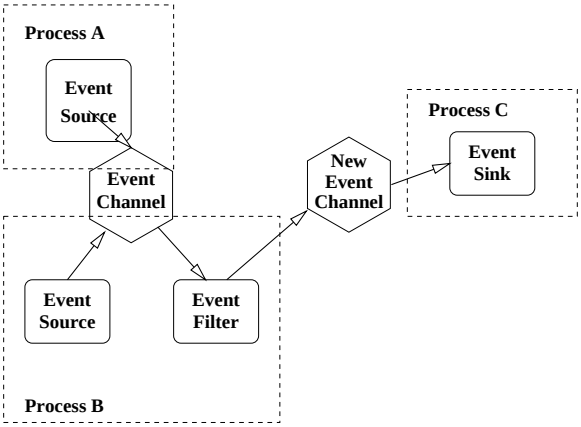

Figure 5: Filter with more than one source.

The event filter can be located on the same node as the event source and is a normal event sink to the original event channel and a normal source to the new, or filtered, event channel. This is a nice solution in that it does not disturb the normal function of the original event channel. However, it fails if there is more than one event source associated with the original event channel. The difficulty is that, as a normal sink, the event filter must live in some specific process. If multiple sources have subscribed to the original event channel and those sources are not co-located, as shown in Figure 5, then there are still raw events traveling over the network from Process A to Process B to be filtered.

The normal semantics of event delivery schemes do not offer an appropriate solution to the event filtering problem. Yet it is important problem to solve because of the great potential for reducing resource requirements if unwanted events can be suppressed. Our approach involves extending event channels with the concept of a *derived* event channel. Rather than explicitly creating new event channels with intervening filter objects, applications that wish to receive filtered event data create a new channel whose contents are derived from the contents of an existing channel through an application supplied derivation function, F. The event channel implementation will move the derivation function F to all event sources in the original channel, execute it locally whenever events are submitted and transmit any event that results in the derived channel. This approach has the advantage that we limit unwanted event traffic (and the associated waste of compute and network resources) as much as possible. Figure 6 shows the logical arrangement of a derived event channel.

3.2 Mobile Functions and the E-code Language

A critical issue in the implementation of derived event channels is the nature of the function F and its specification. Since F is specified by the sink but must be evaluated at the (possibly remote) source, a simple function pointer is obviously insufficient. There are several possible approaches to this problem, including:

- severely restricting F, such as to preselected values or to boolean operators,
- relying on pre-generated shared object files, or
- using interpreted code.

Having a relatively restricted filter language, such as one limited to combinations of boolean operators, is the approach chosen in the CORBA Notification Services[7] and in Siena[1]. This approach facilitates efficient interpretation, but the restricted language may not be able to express the full range of conditions that may be useful to an application, thus limiting its applicability. To avoid this limitation it is desirable to express F in the form of a more general programming language. One might consider supplying F in the form of

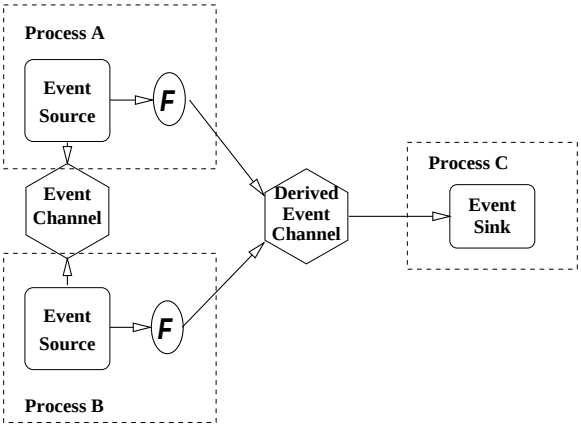

Figure 6: A derived event channel and function F moved to event sources.

a shared object file that could be dynamically linked into the process of the event source. Using shared objects allows F to be a general function, but requires the sink to supply F as a native object file for each source. This is relatively easy in a homogeneous system, but becomes increasingly difficult as heterogeneity is introduced.

In order to avoid problems with heterogeneity one might supply F in an interpreted language, such as a TCL function or Java code. This would allow general functions and alleviate the difficulties with heterogeneity, but it impacts efficiency and requires a potentially large interpreter environment everywhere event channels are used. Given that many useful filter functions are quite simple and given our intended application in the area of high performance computing we rejected these approaches as unsuitable. Instead, we consider these and other approaches as a complement to the methods described next.

The approach taken in ECho preserves the expressiveness of a general programming language and the efficiency of shared objects while retaining the generality of interpreted languages. The function F is expressed in E-Code, a subset of a general language, and dynamic code generation is used to create a native version of F on the source host. E-Code may be extended as future needs warrant, but currently it is a subset of C. Currently it supports the C operators, `for` loops, `if` statements and `return` statements. Extensions to other language features are straightforward and several are under consideration as described in Section 4.

E-Code's dynamic code generation capabilities are based on Icode, an internal interface developed at MIT as part of the 'C project[12]. Icode is itself based on Vcode[5] also developed at MIT by Dawson Engler. Vcode supports dynamic code generation for MIPS, Alpha and Sparc processors. We have extended it to support MIPS n32 and 64-bit ABIs and x86 processors[3]. Vcode offers a virtual RISC instruction set for dynamic code generation. The Icode layer adds register allocation and assignment. E-Code consists primarily of a lexer, parser, semanticizer and code generator.

ECho currently supports derived event channels that use E-Code in two ways. In the first, the event type in the derived channel is the same as that of the channel from which it is derived (the parent channel). In this case, the E-Code required is a boolean filter function accepting a single parameter, the input event. If the function returns non-zero it is submitted to the derived event channel, otherwise it is filtered out. Event filters may be quite simple, such as the example below:

[3] Integer x86 support was developed at MIT. We extended Vcode to support the x86 floating point instruction set (only when used with Icode).

```
extern EChannel EChannel_derive(DExchange de, char *chan_id, char *filter_function);
extern EChannel EChannel_typed_derive(DExchange de, char *chan_id, char *filter_function,
                                      IOFieldList field_list, DEFormatList format_list);
```

Figure 7: Derived event channel API.

```
{
    if (input.level > 0.5) {
        return 1; /* submit event into derived channel */
    }
}
```

When used to derive a channel, this code is transported in string form to the event sources associated with the parent channel, is parsed and native code is generated at those points. The implicit context in which this code evaluated is a function declaration of the form:

`int f(`⟨*input event type*⟩ `input)`

where ⟨*input event type*⟩ is the type associated with the parent channel.[4] The API for channel derivation is shown in Figure 7. Once derived, the created channel behaves as a normal channel with respect to sinks. It has all of the sources of the parent channel as implicit sources, but new sources providing unfiltered events can also be associated with it.

While this basic support for event filtering is a very powerful mechanism for suppressing unnecessary events in a distributed environment, ECho also supports derived event channels where the event types associated with the derived channel is not the same as that of the parent channel. In this case, E-Code is evaluated in the context of a function declaration of the form:

`int f(`⟨*input event type*⟩ `input,` ⟨*output event type*⟩ `output)`

The return value continues to specify whether or not the event is to be submitted into the derived channel, but the differentiation between input and output events allows a new range of processing to be migrated to event sources.

One use for this capability is remote data reduction. For example, consider event channels used for monitoring of scientific calculations, such as the global climate model described in [9]. Further, consider a sink that may be interested in some property of the monitored data, such as an average value over the range of data. Instead of requiring the sink to receive the entire event and do its own data reduction we could save considerable network resources by just sending the average instead of the entire event data. This can be accomplished by deriving a channel using a function which performs the appropriate data reduction. For example, the following E-Code function:

```
{
    int i;
    int j;
    double sum = 0.0;
    for(i = 0; i<37; i= i+1) {
        for(j = 0; j<253; j=j+1) {
            sum = sum + input.wind_velocity[j][i];
        }
    }
    output.average_velocity = sum / (37 * 253);
    return 1;
}
```

performs such an average over atmospheric data generated by the atmospheric simulation described in [9], reducing the amount of data to be transmitted by nearly four orders of magnitude.

[4] Since event types are required, new channels can only be derived from typed channels.

3.3 Generation and Run Time

The benefits which might result from the use of derived event channels can vary significantly depending upon their rejection rate or amount of data reduction. With gains in reduced network usage come additional costs in the form of generating and executing the derivation functions. Those costs vary with the nature of the code as well, but it's informative to examine the costs for some examples to determine their range.

For example, doing dynamic code generation on the first filter example in Section 3.2 required 2.3 milliseconds on a Sun Sparc Ultra 1/140, resulting in the generation of 21 Sparc instructions that execute in about 3 microseconds. In comparison, simply transmitting the unfiltered event costs the sender approximately 5 microseconds[5]. While any at-source event rejection will reduce network traffic, the timings demonstrate that executing the filter function can be cheaper for the sender than the network operation. Thus, (depending upon the rejection rate of the filter) at-source filtering may actually reduce the computational demands for the sender as compared to the unfiltered situation.

The second example is more complex and requires 7 msec to do code generation and 1.6 msec to execute. In comparison, simply transmitting the original unreduced event costs the application only 1.1 msec. Unlike the previous example where the filter function was cheaper than transmission, there are no circumstances in which this particular data reduction will reduce the computational demands of the sender. In essence, deriving such a channel is moving the averaging calculation from the event receiver to the sender, resulting in a significant reduction in network traffic but increasing the workload of the sender. Whether or not this trading of resources is worthwhile is an application-specific decision. The contribution of derived event channels is that it facilitates such tradeoffs.

4 Ongoing Work

Derived event channels are a recent result and are subject to continuing improvement as we gain experience with their use in real applications. However, some areas for future work are already apparent:

Parameterized Filters Currently derivation functions are simple functions of their input events. However, there are some obvious ways where more powerful functions could be valuable. Consider the situation where a sink wants a filter function based on values which change (hopefully more slowly than the event stream they are filtering). A simple example might occur in a distributed virtual reality application using event channels to share participant location information. Rather than sending location information as fast as the network allows, a more intelligent system might use derived event channels to turn down the update rate when the participants are not in sight of each other or merely distant. However, these conditions obviously change over time. One could periodically destroy and re-derive channels with updated filter functions, but a more straightforward approach would be to associate some state with a derivation function and allow it to be updated by the originator.

Visibility into Source Process [4] anticipates using the functionality offered by derived event channels for program steering. Currently external program steering typically requires monitoring events to be delivered to an external process. There they are interpreted and actions can be taken to affect the state of the application being steered. Like filter functions, steering decision functions may be as simple as comparing an incoming value with a threshold and steering the program if the threshold is exceeded. As a result, program steering requires a minimum of a network round-trip, and it is an asynchronous process. If the steering

[5] Transmission time derived from measured bandwidth for appropriately sized message sends on the host machine across 100Mbps ethernet LAN.

functions could be migrated into the steered application, much like we are moving filter functions to event sources, steering latencies could be reduced by several orders of magnitude and undertaken synchronously. When program steering is performed by a human-in-the-loop system, latency and synchronicity are of little importance. But steering can also be performed algorithmically and holds significant potential in terms of allowing programs, system abstractions and even operating systems to adapt to changing usage and resource situations[13, 11]. Such adaptions must be rapid and often must be synchronous in order to be valid.

However, doing program steering in a derivation function requires that the function have visibility into application program to call functions or affect values there. How this can be cleanly accomplished is an open question, but we anticipate mechanisms for associating a "context" with an event channel. The context would specify program elements which might be accessible to derivation functions. In this way, visibility into the sending application would be controlled by the sender but the use and updating of visible elements would be specified by the receiver through derivation function.

Function Analysis There is also potential benefit in specializing and analyzing the derivation functions. In particular, an event source clearly has an interest in understanding the nature of the code it agrees to execute for a derivation function. While the simplest functions may be safe in the sense that they have no visibility into source's environment, they may walk off the end of event arrays or contain infinite loops. However, in this environment we know the size and extent of all data types. We can easily generate array bounds checks and can consider analyzing the specified functions for computational complexity and termination. An event source on a highly loaded machine might be given the option of rejecting a derivation that would increase its workload. More generally, since derived event channels always represent some tradeoff between compute and network resources, we can consider creating a more generalized mechanism through which tradeoff decisions can be made in the context of dynamic feedback from resource monitoring. Such a system could greatly ease the process of building applications which perform robustly in today's dynamic network environments.

Optimization The dynamic code generation tools used in Derived Event Channels generate code quickly, but the generated code is not well optimized. For example, the filter whose dynamic code generation was described above to have resulted in 21 Sparc instructions can be correctly performed in 10 instructions. Vcode includes a peephole optimizer for the Sparc architecture, but the generated code could clearly benefit from increased optimization. However, optimization is a significant source of complexity and compile-time cost in modern compilers. Whether the efficiency gains of additional optimization are significant enough to outweigh their increased costs is an avenue for future research.

5 Conclusion

We believe that event filtering through the derived event channel model presents an intuitive and natural mechanism for limiting the unnecessary network traffic associated with undesirable events. The use of dynamic code generation allows computationally complex filter computations to be expressed without the overhead of interpreted code.

Acknowledgments

This work was performed with the support of NSF CISE grant 9720167. Matt Sanders contributed to an early prototype of derived event channels. Karsten Schwan provided valuable counsel in the development of the concepts presented in this paper.

References

[1] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design of a scalable event notification service: Interface and architecture. Technical report, Department of Computer Science, University of Colorado at Boulder, August 1998. Technical Report CU-CS-863-98.

[2] Greg Eisenhauer. Portable self-describing binary data streams. Technical Report GIT-CC-94-45, College of Computing, Georgia Institute of Technology, 1994. *(anon. ftp from ftp.cc.gatech.edu)*.

[3] Greg Eisenhauer, Beth Schroeder, and Karsten Schwan. Dataexchange: High performance communication in distributed laboratories. *Journal of Parallel Computing*, (24), 1998.

[4] Greg Eisenhauer and Karsten Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, August 1998.

[5] Dawson R. Engler. Vcode: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of ACM SIGPLAN'96 Conference on Programming Language Design and Implementation*, 1996.

[6] Object Management Group. *CORBAservices: Common Object Services Specification*, chapter 4. OMG, 1997. http://www.omg.org.

[7] Object Management Group. Notification service. http://www.omg.org, Document telecom/98-01-01, 1998.

[8] JavaSoft. Distributed event specification. http://www.javasoft.com, 1998.

[9] Thomas Kindler, Karsten Schwan, Dilma Silva, Mary Trauner, and Fred Alyea. A parallel spectral model for atmospheric transport processes. *Concurrency: Practice and Experience*, 8(9):639–666, November 1996.

[10] Balachander Krishnamurthy and David S. Rosenblum. Yeast: A general purpose event-action system. *IEEE Transactions on Software Engineering*, 21(10):845–857, October 1995.

[11] B. Mukherjee and K. Schwan. Implementation of scalable blocking locks using an adaptive threads scheduler. In *International Parallel Processing Symposium (IPPS)*. IEEE, April 1996.

[12] Massimiliano Poletto, Dawson Engler, and M. Frans Kaashoek. tcc: A template-based compiler for 'c. In *Proceedings of the First Workshop on Compiler Support for Systems Software (WCSSS)*, February 1996.

[13] Daniela Ivan Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On adaptive resource allocation for complex real-time applications. In *18th IEEE Real-Time Systems Symposium, San Francisco, CA*, pages 320–329. IEEE, Dec. 1997.