Profile-Directed Optimization of Event-Based Programs *

Mohan Rajagopalan Saumya Debray
Department of Computer Science
University of Arizona
Tucson, AZ 85721
{mohan, debray}@cs.arizona.edu

Matti A. Hiltunen Richard D. Schlichting
AT&T Labs-Research
180 Park Avenue
Florham Park, NJ 07932
{hiltunen, rick}@research.att.com

ABSTRACT

Event-based systems provide a simple way to create flexible, extensible, and customizable system architectures and give a "user-driven" feel to the system. However, the indirect coupling between the raising and handling of events introduces a number of overheads into the system. Such overheads can be surprisingly large, and are especially significant in small mobile systems. This paper describes a framework for profile-guided optimization of event-based systems. Experiments using our approach on two different event-based systems, Cactus and X-windows, indicates that it can achieve significant reductions in event handling overheads and lead to considerable improvements in overall system performance.

1. INTRODUCTION

Events are increasingly being used as a fundamental abstraction for writing programs in a variety of contexts. They are used to structure user interaction code in GUI systems [16, 7], form the basis for configurability in systems to build customized distributed services and network protocols [3, 8, 14], are the paradigm used for asynchronous notification in distributed object systems [17], and are advocated as an alternative to threads in web servers and other types of system code [18, 21]. Even operating system kernels can be viewed as event-based systems, with the occurrence of interrupts and system calls being events that drive execution. The rationale behind using events is multidimensional. Events are asynchronous, which is a natural match for the reactive execution behavior of GUIs and operating systems. Events also allow the modules raising events to be decoupled from those fielding the events, thereby improving configurability. In short, event-based programming is generally more flexible and can often be used to realize richer execution semantics than traditional procedural or thread-oriented styles.

Despite these advantages, events have the potentially serious drawback of extra execution overhead due to the indirection between modules that raise and handle events [4, 13]. Typically, there is a registry that maps an event to a collection of handlers to be executed when the event occurs. Because these handlers are not known statically—and may in fact change dynamically—they are invoked indirectly. Depending on the system, the number and type of the arguments passed to the handler may also not be known, requiring argument marshaling. Finally, there may be repeated work, e.g., initialization or checking of shared data structures, across multiple handlers for a given event. All these extra costs can be surprisingly high—our experiments indicate that they can account for up to 20% of the total execution time in some scenarios.

This paper describes a collection of static optimizations designed to reduce the overhead of event-based programs. Our approach exploits the underlying predictability of many event-based programs to generate an *event profile* that is conceptually akin to path profiles through the call graph of the program. These profiles are then used to identify commonly encountered events and their handlers, as well as the collection of handlers associated with each event and the order in which they are invoked. This information is then used to optimize event execution by, for example, merging handlers and chaining events. The techniques are specific to event-based programs, since standard optimization techniques are largely ineffective in this context. For example, conventional static analysis techniques cannot generally discover the connections between events and handlers, let alone optimize away the associated overheads. Dynamic optimization systems such as Dynamo [1] can be used in principle, but they focus primarily on lightweight optimizations such as improving locality and instruction-cache usage in an effort to keep runtime overheads low. In contrast, the optimizations we consider are substantially more heavyweight, and—in the context of event-based programs—offer correspondingly greater benefits. Our techniques are specifically designed to improve execution on small mobile devices, where resource constraints make any reduction in overhead valuable.

The remainder of the paper is organized as follows. Section 2 describes a general model for event-based programs. This is followed in section 3 by a description of our approach to optimizing such programs, including our profiling scheme and the collection of optimization techniques based on these profiles. Section 4 gives experimental results that demonstrate the potential improvements for three different examples. The first two, a video application and a configurable secure communication service, are built using Cactus, a system for constructing highly configurable distributed services and network protocols that supports event-based execution [11, 9]. The third is client side tools that use X Windows, a popular system for building GUIs [16]. This is followed by discussions of possible extensions in section 5 and related work in section 6. Finally, section 7 offers conclusions.

*The work of S. Debray was supported in part by the NSF under grants CCR-0073394, EIA-0080123, and CCR-0113633. The work by others supported in part by the DARPA under grant N66001-97-C-8518 and by the National Science Foundation under grant ANI-9979438.

2. EVENT-BASED PROGRAMS

While event-based programs differ considerably depending on the specifics of the underlying programming model and notation, their architectures have a number of broad underlying similarities. Because of this, the optimizations described in this paper are generally applicable to most such systems. This section presents a general model for event-based systems in order to provide a common framework for discussion. As examples, we describe how both Cactus and the X Windows system map into the model.

2.1 Components

Our general model consists of three main components: *events*, *handlers* that specify the reaction to an event, and *bindings* that specify which handlers are to be executed when a specific event occurs.

Events. Events abstract the asynchronous occurrence of stimuli that must be dealt with by a program. Mouse motion, button click, and key press are examples of such events in a user interface context, while receiving a packet from the network and message passing are examples in a systems context. In addition to such *external events*, an event-based program may use *internal events* that are generated and processed within the program. The set of events used in the event system may be fixed or the system may allow programs to define new events. Basic events may be composed into *complex events*. For example, two basic button click events within a short time period can be defined to constitute a double-click event.

Handlers. Handlers direct the response of the program to event-based stimuli. Specifically, a handler is a section of code that specifies the actions to be performed when a given event occurs. Typically, handlers have at least one parameter, the event that was raised; other parameters may be passed through variable argument lists or through shared data structures. The decoupling provided by the event mechanism allows handlers to be developed independently from other handlers in the system.

Bindings. Bindings determine which handlers are executed when a specific event occurs. The binding between an event and a hander is often provided using some type of *bind* operation, although the binding may also be predefined and fixed. Most systems allow multiple handlers to be bound to a single event and a handler to be bound into more than one event. An event is ignored if no handlers are bound to the event. The execution order of multiple handlers bound to the same event may be important. Bindings may be *static*, i.e., remain the same throughout the execution of the program, or *dynamic*, i.e., may change at runtime. Figure 1 illustrates bindings.

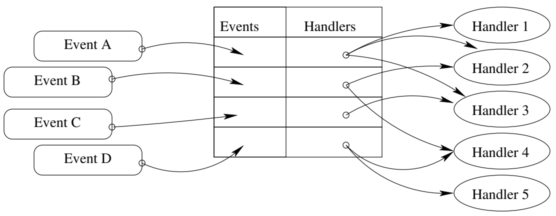

Figure 1: Event Bindings

Bindings are maintained in a registry that maps each event to a list of handlers. The registry may be implemented as a shared data structure like the table shown in the figure, or each list may be maintained as a part of an event data structure. For distributed systems where handlers may be on distinct physical machines, the registry may be implemented using either a centralized or decentralized approach.

2.2 Execution

The handlers bound to an event are executed when the event occurs. An event may occur because the program receives some external stimuli (external event) or because some program component raises the event (internal event). An execution environment or runtime system is typically responsible for detecting or receiving external stimuli and activating the corresponding events. As a result, we say these events are raised implicitly, whereas events directly activated by a program component are raised explicitly. *Timed events* are events that are activated at a specified time or after a specified delay. These can be either internal or external events.

We identify two major types of event activation: *synchronous activation* and *asynchronous activation*. With synchronous activation, the specified handlers are executed to completion before the activator continues execution. With asynchronous activation, the activator continues execution without any guarantees as to when the handlers are executed. The different types of event activation have specific uses in event-based systems. Synchronous activation can be used for internal events when the event activator needs to know when the processing of the message has completed before continuing its own processing. Synchronous activation can be used for external events when the runtime system needs to ensure that such events are executed sequentially without interleaving. Asynchronous activation can be used when none of these requirements apply.

The overall picture of the event-based program to be optimized then consists of a program that reacts to stimuli from its environment, such as user actions or messages. These stimuli are converted into events. Each event may have multiple handlers bound to it and handlers may activate other events synchronously or asynchronously. Thus, the occurrence of an event may lead to the activation of a chain of handlers and other events and, in turn, their handlers. Events can also be generated by the passage of time (e.g., timeouts). The type of event activation has implications on our optimization techniques. For example, since the handlers for a synchronous activation are executed when the event is raised, an optimization that replaces the activation call with calls to the handlers bound to that event at this time results in a correct transformation. Similarly, it is easy to see that sequences of or nested synchronous activations can be readily optimized. The specific optimization techniques and their limitations are discussed below in section 3.

2.3 Example Systems

Cactus. Cactus is a system and a framework for constructing configurable protocols and services, where each service property or functional component is implemented as a separate module [9]. As illustrated in figure 2, a service in Cactus is implemented as a *composite protocol*, with each service property or other functional component implemented as a *micro-protocol*. A customized instance of the composite protocol is constructed simply by choosing the appropriate set of micro-protocols. A micro-protocol is structured as a collection of *event handlers* that correspond to the handlers in our general event-based model. A typical micro-protocol consists of two or more event handlers. Events in Cactus are user-defined. A typical composite protocol uses 10-20 different events consisting of few external events caused by interactions with software outside the composite protocol and numerous internal events used to structure the internal processing of a message or service request. Each event typically has multiple event handlers. As a result, Cactus composite protocols often have long chains of events and event handlers activated by one event. Section 4 gives concrete

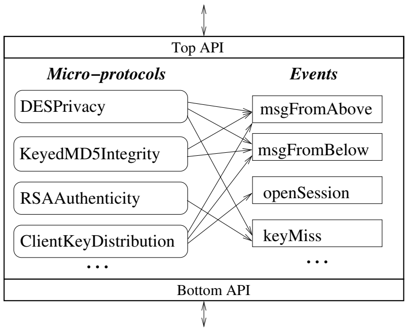

Figure 2: Cactus Composite Protocol

examples of events used in a Cactus composite protocol.

The Cactus runtime system provides a variety of operations for managing events and event handlers. In particular, operations are provided for binding an event handler to a specified event (*bind*) and for activating an event (*raise*). Event handler binding is completely dynamic. Events can be raised either synchronously or asynchronously, and an event can also be raised with a specified delay to implement time-driven execution. The order of event handler execution can also be specified if desired. Arguments can be passed to handlers in both the bind and raise operations. Other operations are available for unbinding handlers, creating and deleting events, halting event execution, and canceling a delayed event. Handler execution is atomic with respect to concurrency, i.e., a handler is executed to completion before any other handler is started unless it voluntarily yields the CPU. Cactus does not directly support complex events, but such events can be implemented by defining a new event and having a micro-protocol raise this event when the conditions for the complex event are satisfied.

The X Window system. X is a popular GUI framework for Unix systems. The standard architecture of an X based system is shown in figure 3. The X server is a program that runs on each system supporting a graphics display and is responsible for managing device drivers. Application programs, also called X clients, may be local or remote to the display system. X servers and X clients use the X-protocol for communication. X clients are typically built on the Xlib libraries using toolkits like Xt, GTK, or Qt. X clients are implemented as a collection of *widgets*, which are the basic building blocks of X applications.

An X event is defined as "a packet of data sent by the server to the client in response to user behavior or to window system changes resulting from interactions between windows" [16]. Examples of X events include mouse motion, focus change, and button press. These events are recognized through device drivers and relayed to the X server, which in turn conveys them to X clients. The Xlib framework specifies 33 basic events. X clients may choose to respond to any of these based on event masks, which they specify at bind time. Events are also used in communication between widgets. Events may arrive in any order and they are queued by the

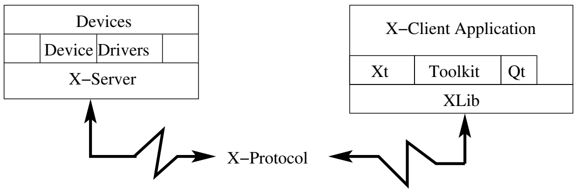

Figure 3: Architecture of X Window Systems

X client. The event activation in X is similar to the synchronous activation in our general model.

The X architecture has a number of different methods for handling events: *event handlers*, *actions*, and *callbacks*. Although all of these map into the handlers in our general model, they have significant differences. For example, while actions can be specified for an X client, event handlers and callbacks are specified for each widget in the X client. While each callback is bound to a specific callback name and all the callbacks associated with this name are executed when the specific callback name is called, an event handler can be bound to be executed when *any* of the events that are specified using an event mask occur. Actions provide an additional level of indirection, where a mapping is created first between an event and the action name and then between the action name and the action procedure to be called. In addition to these three, X has a number of other mechanisms that can be broadly classified as event handling, namely *timeouts*, *signal handlers*, and *input handlers*. Each of these mechanisms allows the program to specify a procedure to be called when the specific condition occurs. For all of these handler types, X provides operations for registering the handlers and activating them.

3. OPTIMIZATION APPROACH

Compiler optimizations are based on being able to statically predict some aspects of a program's runtime behavior, either via invariants that always hold at runtime (e.g., based on dataflow analyses), or that are likely to hold (e.g., based on execution profiles). Event-based systems, by contrast, are largely unpredictable in their runtime behavior due to unpredictabilities associated with the behavior of their external environment, e.g., the user's actions. We have found, however, that in practice, there is a significant amount of predictability in their internal behavior that can be exploited for optimization purposes. This predictability occurs at two levels. At the event level, certain sequences of events can be found to occur in all (or most) system executions. At the handler level, there is often more than one handler bound to a specific event, and all of these handlers are executed in sequence each time the event occurs.

We identify predictable aspects of the behavior of event-based systems using event and handler profiling. This section describes our profiling techniques and the optimizations we carry out based on these profiles.

3.1 Event Profiling

We identify static optimization opportunities in an event-based program using a event and handler execution profiles. We first identify commonly occurring event sequences by instrumenting the event system to log an entry each time an event occurs, indicating the event being raised and whether it is being raised synchronously or asynchronously. We use the resulting event profiles to identify frequently invoked event handlers, add instrumentation code to each such handler, and log entries each time the handler is invoked, thereby obtaining handler profiles. Profiling is done to one program—and for configurable programs, one program configuration—at a time. At present, the event framework is instrumented by hand, but this can easily be automated using well-understood techniques [2]. The analysis and optimizations are currently performed off-line after the program to be optimized is executed enough times. On-line analysis, and potentially optimization, are potential extensions to this work and are discussed in section 5.

The profiling algorithm takes the event trace generated by the instrumented event framework and generates an *event graph*, which summarizes the event sequences in the trace. There is an edge from node A to node B in the event graph if event A is ever followed immediately by event B in the event trace. Each edge (A, B) has an

```
EventGraph = ∅;
prev_event = eventTrace→firstEvent;
while not (end of eventTrace) {
    event = eventTrace→nextEvent;
    if (prev_event,event) not in EventGraph {
        EventGraph += (prev_event,event);
        EventGraph(prev_event,event)→weight = 1;
    } else
        eventGraph(prev_event,event)→weight++;
    prev_event = event;
}
```

Figure 4: *GraphBuilder* **algorithm.**

associated weight indicating how many times the sequence $\langle A, B\rangle$ appeared in the trace. The algorithm used to generate the event graph is presented in figure 4. Note that in the event trace, if an event A is followed immediately by an event B that was raised synchronously, then we can infer that execution of B follows A sequentially. However, if B was raised asynchronously, then the fact that it follows A in the event trace may be pure happenstance: we cannot conclude that A had any role in raising B. For example, B may be the result of a timeout from an earlier event completely unrelated to A.

The event graph is used as the starting point for the analysis that identifies predictable event and handler sequences. Commonly occurring event sequences can be easily identified in the event graph through edge weights. Given an event graph G and a threshold η, we define an *event path* of weight η in G as a path such that no edge on the path has edge weight less than η. To simplify the algorithm, we first discard from the event graph edges whose weights are below the threshold η: this produces a *reduced event graph*, from which we extract event paths. Each event path indicates a frequent sequence of events and hence represents a candidate for optimization. The remainder of this discussion focuses on event paths unless otherwise mentioned. Notice that the event paths so constructed are not quite the same as hot path profiles: the reason we do not use path profiling at the level of events is that path profiles tend to be large and expensive to compute [12, 23], and the results we see experimentally using the approach described above have been adequate for the optimizations we implement.

Since an event may have multiple handlers that are executed in sequence each time the event occurs, a handler level profiling is required to identify predictable sequences of handler activation (because of the decoupling between events and their handlers, knowing the events that occur does not, in itself, tell us about the handlers that are activated). The event paths in the event graph identify the most promising events for handler level profiling. The handlers for the nodes in each event path are instrumented and we construct another graph, the *handler graph*, that forms the basis for optimization. The profiling and graph construction for handlers is carried out in the same way as before.

Figure 5 shows the event graph for a video player application implemented on top of a configurable transport protocol CTP built using Cactus [22]; details are given in section 4.2 (the bold edges are discussed later, in Section *3.2.1*). The event paths in this graph can be grown by iteratively reducing the threshold η. Figure 6 shows the corresponding reduced event graph for $\eta = 300$.

3.2 Optimization Techniques

Once we have identified the most frequent event and handler sequences, the optimizations are performed based on the handler graph. Our goal is to eliminate:

1. Marshalling overheads for event raises.

2. Indirect function call and variable argument passing costs.

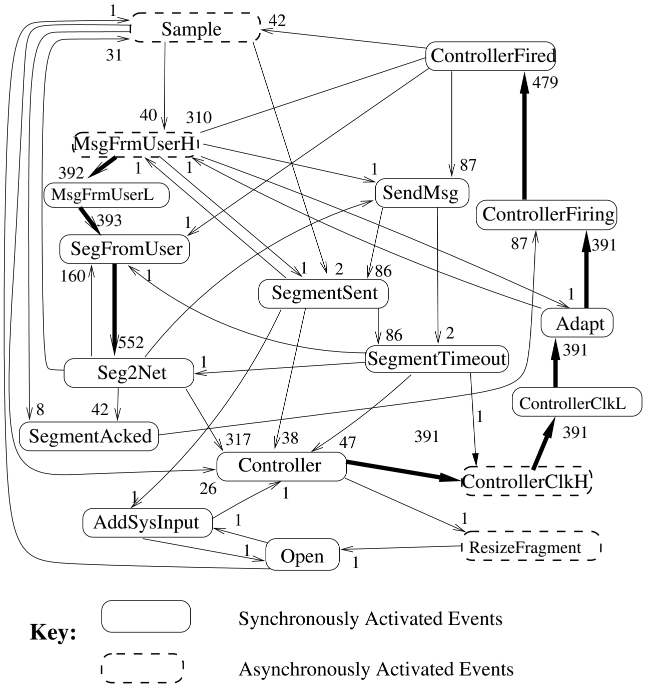

Figure 5: Event Graph Generated from Video Player

3. State maintenance (synchronization and locking) costs for global variables.

4. Redundant initializations and fragments for events with multiple handlers.

In the case of synchronous events we also expect to observe event sequences that can be *chained* together. Elimination of indirect function calls increases the potential for value based optimizations such as constant propagation. Another option we have explored is inlining code for raising popular events. This section takes a look at our different optimizations. Broadly, these can be classified as graph and compiler optimizations.

3.2.1 Event Graph Optimizations

Event graph optimizations try to reduce the costs associated with interactions between events and handlers in the system, by reducing the number of handler activations along common event paths. This is done by reducing the number of nodes in an event graph and

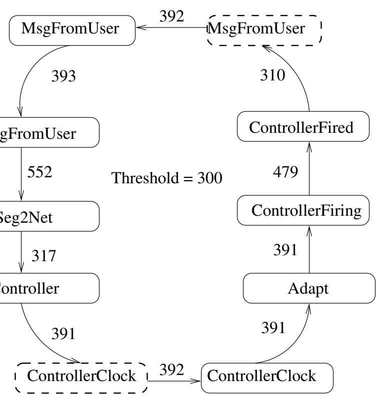

Figure 6: Reduced Event Graph

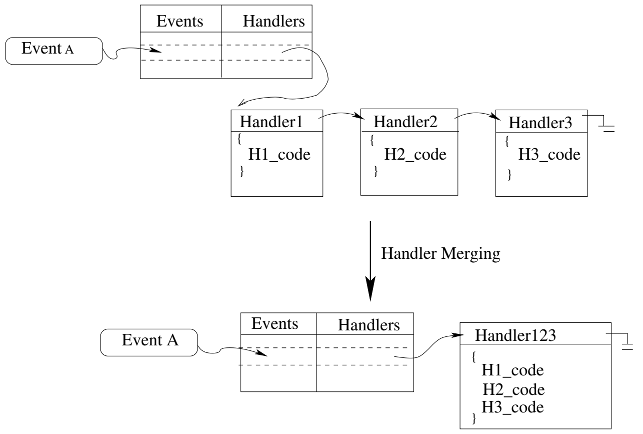

Figure 7: Handler Merging

generating simpler collapsed graphs, as well as by merging handler nodes to create "super-handlers" for events and chains.

Handler Merging: In case of events with multiple handlers, the handler graph shows a sequence of contiguous nodes. The event system is responsible for issuing calls to all handlers bound to the event. References to handlers are stored as function pointers in a list associated with the event, and each raise operation for the event translates into a sequence of indirect function calls. There are two sources of overhead here: the cost of an indirect call, and—since in general the identities and the number of arguments taken by the handlers for an event are not statically known—a cost associated with argument marshaling and unmarshaling. However, we can use the handler graph obtained from our event handler profiles to identify the sequence of handlers activated when an event is raised. Given this information, a simple approach for dealing with this overhead is to merge all the handlers associated with an event into a single large handler. On the handler graph, this translates to collapsing all handler nodes for a given event into a single *super-handler node*. The immediate savings from this transformation is the reduction in the number of indirect function calls. Figure 7 shows the effect of this optimization. Further savings then result from the application of standard compiler optimizations, such as common subexpression elimination and dead-code elimination , on the super-handler code.

An important point to note in this context is that some event systems such as Cactus allow event bindings to change dynamically. We need to account for such changes and ensure that the correctness is preserved by our optimization even if there are such dynamic changes. Our experiments indicate that dynamic changes to the set of handlers for an event are rare: we keep track of whether or not there have been any changes to the set of handlers for an event, and drop back into the original unoptimized code for the event if a change is detected. See section 3.3 for more details.

Raise operations are typically generic, in the sense that they can raise any arbitrary event. Because of this, they incur overheads due to argument marshaling, indirect invocation of handlers, and state maintenance. The number of indirect raises—and hence their total cost—can be reduced using super-handlers, as discussed above. However, super-handlers are still invoked indirectly, and so incur some residual cost. These costs can be reduced by replacing the raise operation with a direct call to the super-handler based on profile information. This, in turn, opens up the possibility of inlining the function call into the call site, as discussed below.

Event Chains and Subsumption. The unpredictable nature of events may suggest, superficially, that different events are largely independent of each other. However, our experiments indicate that very often there are significant correlations between different events, of the form, *"Event B always follows Event A."* This leads to commonly occurring sequences of events, which we term event chains, that are candidates for optimization. An *event chain* is defined to be a path in the event graph

$$a_0 \to a_1 \to \cdots \to a_n \qquad [n > 0]$$

satisfying the following:

(i) each vertex a_i in the chain (except possibly for the last vertex, a_n) has exactly one successor edge corresponding to a synchronous event activation; and

(ii) the edge $a_i \to a_{i+1}$ corresponds to a synchronous event activation.

The intuition behind event chains is that they denote sequences of event activations that we can guarantee will occur if the event at the head of the chain occurs. The reason we ignore event graph edges corresponding to asynchronous raises is that, as mentioned earlier, if an event A is followed by an asynchronously raised event B, we cannot infer that B was raised by A. Event chains can arise due to two reasons. First, the configuration of the event system may be such that a particular set of events is raised in sequence under the appropriate circumstances (we see this situation commonly in the X-windows system). Second, the handlers of one event may (synchronously) raise other events. Two examples of event chains are shown in figure 5 as sequences of bold edges.

We optimize event chains in two ways. First, we can generalize the notion of handler merging (section *3.2.1*) to span event boundaries. The effect is to combine all of the handlers for all of the events in the chain into a single handler. This has the effect of avoiding the runtime cost of multiple handler invocations. If this is not possible for some reason (see below), we can optimize the handlers for events later in the chain knowing that the handlers for events earlier in the chain have been executed. This may allow us to eliminate some redundant work across handlers.

Inter-event handler merging is not carried out if any of the events in an event chain is an asynchronous event (or, as a special case, a timed event, see section 2.2). This is necessary in order to preserve the observable behavior of the system with respect to the timing semantics of the system. For example, suppose that event A, signifying "data transfer initiated," is always followed by event B, signifying "data transfer completed," but that B occurs (at least) some fixed time after A. In this case, B is raised as an asynchronous event, and so is not subjected to handler-merging. Note that this also maintains the threading semantics of the system. In case of synchronously activated events, the activator thread services all of its handlers, whereas in the case of asynchronously activated events this may not be the case. Maintaining asynchronous events in the system replicates this behavior. It should be noted that even though handler merging is not carried out in such cases, the event chains can still be optimized to some extent based on knowledge about the context in which later events can be raised, e.g., in this example we know that when event B is raised, the event A must have been raised, and handled, previously.

There is an important special case of event chain optimization where the handlers of one event raise other events synchronously. This leads to event chains where handlers for an event B may be embedded in the handlers for the parent event A. An example of this is found in the video player example shown in figure 5, and is described by figure 8, which focuses on two distinct events, `SegFromUser` and `Seg2Net`, from figure 5; the former is shown

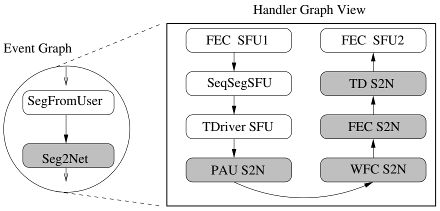

Figure 8: Subsumption of Events

unshaded, the latter shaded gray. The relevant portion of the corresponding handler graph is shown on the top right of figure 8, with handlers arising from the actions of `SegFromUser` shown unshaded and those arising from `Seg2Net` shown shaded. The proper nesting of the shaded handler sequence within the unshaded sequence indicates that `Seg2Net` is called *synchronously* by the handlers of `SegFromUser`. In other words, if the event `Seg2Net` is raised from within a handler for `SegFromUser`, the latter will wait until the handling of `Seg2Net` has been completed, at which point control will return to the handler for `SegFromUser`. In this case, we can simply *subsume* the handler for `Seg2Net` into that for `SegFromUser`, thereby eliminating the synchronous event raise between them.

Figure 9 shows the effects of event subsumption for the events shown in Figure 8. The event `SegFromUser` has four handlers: `FEC-SFU1`, `SeqSegSFU`, `TDriver-SFU`, and `FEC-SFU2`, while `Seg2Net` has four handlers `PAU-S2N`, `WFC-S2N`, `FEC-S2N`, and `TD-S2N`. `TDriver-SFU` synchronously raises the event `Seg2Net`, which causes the execution of its handlers, after which control returns to handling `SegFromUser` and causes the last handler, `FEC-SFU2`, to be executed. Without event subsumption, the best we would be able to do is to merge the handlers for `SegFromUser` to create a super-handler for it, and similarly for `Seg2Net`. This would still incur the overhead of an event raise of `Seg2Net` from `TDriver-SFU`. However, the synchronous nature of this raise allows us to optimize this code so that the event raise of `Seg2Net` within the superhandler for `SegFromUser` is replaced by the handler code for `Seg2Net`, as illustrated in Figure 9.

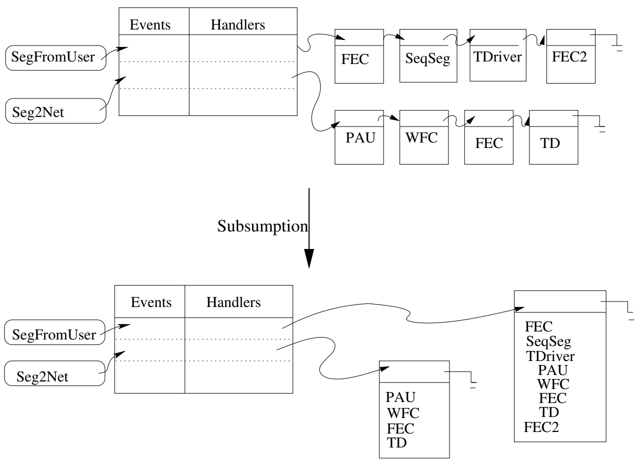

Figure 9: Effect of Event Subsumption

3.2.2 Compiler Optimizations

The super-handlers resulting from event graph optimization have the effect of bringing together code that was, prior to optimization, scattered across the program over a number of different handler routines. As a result, they become amenable to further optimization via standard compiler optimization techniques. We mention below some of these optimizations that tend to be especially useful in this context.

Function Inlining. Since most event handlers tend to be relatively small in size, function inlining—applied aggressively along, and restricted to, frequently executed event paths—is very effective in reducing the overhead of function calls without substantial growth in code size. Additionally, a secondary benefit of inlining is the reduction in the cost associated with supporting variable numbers of arguments to a function. Typically event-based systems do not regulate the types and number of parameters for a handler. This leads to the use of variable argument lists in handlers. Inlining handler code allows us to eliminate the overheads associated with argument marshaling and unmarshaling.

Constant Propagation and Dead Code Elimination. Function inlining makes it straightforward to propagate information about constant arguments from the call site into the inlined code. This in turn exposes the potential for constant propagation optimizations to be applied to the super-handler. For example, conditionals that test the value of one or more arguments can be eliminated if the values of the corresponding arguments are known at the call site where inlining is carried out. As a specific case, this allows us to take the code for a handler that could be invoked by multiple events (e.g., Handler 4 in figure 1), and create a customized version of the (inlined) handler corresponding to a frequently encountered event.

General purpose handlers may be shared amongst different events. Typically such handlers contain conditional statements for deciding the nature of handling. Constant propagation, together with optimizations it enables such as the elimination of conditional branches whose outcomes become known, can cause code to become dead or unreachable. The elimination of such useless code further improves handler and system performance.

Redundant Code Elimination. Handlers are not required to know about the behavior of other handlers in the system. Because of this, they usually do not make assumptions about the actions that may have been carried out by other handlers. This can lead to redundant code in the system. For example, a handler may evaluate expressions that have already been evaluated by preceding handlers in an event chain. Another common example is that of managing system state: since the system state may change because of handlers, in a multithreaded context handlers typically start by updating their state from the global system state, and commit new system state at the end. In an event chain, this can cause redundant updates of the handler state. Such redundant operations can be identified in the super-handler code and are target for this optimization.

3.3 Dealing with the Unexpected

A limitation of profiling is the fact that it will typically not explore all possible program execution paths. For example, the program may choose a completely new path when given an input that was not used in the profiling test runs. Since the optimized program must operate in the same way as (except faster than) the corresponding unoptimized version, we must have a mechanism for ensuring that the optimized super handlers are only used when the unoptimized program would have taken the corresponding sequence of handlers. We solve this problem by bypassing the optimized super handlers when the program would not execute this sequence of handlers.

The specific solution is based on keeping track of event bindings in the event framework. We associate a flag with each event path. When the bindings for any event along the path change, it gets re-

Frame rate	Total Execution Time (sec)			Event Handler Time (sec)		
	Orig. (T_0)	Opt. (T_1)	T_1/T_0 (%)	Orig. (T_0)	Opt. (T_1)	T_1/T_0 (%)
10	43.1	41.9	97.2	2.3	0.9	39.1
15	30.9	30.3	98.0	1.6	0.6	37.5
20	24.5	22.1	90.2	1.5	0.5	33.3
25	23.9	21.3	89.1	1.5	0.5	33.3

Key: Orig: Original program; Opt: Optimized program

Figure 10: Video player optimization results.

flected in the flag. This flag is checked before executing the super handler for the path. If the flag is *set*, we fall back to the handlers in the original version of the program. This ensures that the optimized version behaves in the same way as the original. Note that as a result, this optimization technique results in a larger code size since now the event-based program must contain both the super handlers as well as the original unoptimized handlers. However our experiments indicate that this increase is small, typically under 1.5% (see Section 4.2).

The new program can be *re-optimized* as before by continuous profiling. A more efficient solution would be to dynamically rewrite super handlers to reflect the changes in binding. We plan to implement this dynamic configurability as future work. We discuss proposed solutions and provide more details in section 5

4. EXPERIMENTAL RESULTS

4.1 Overview

We ran experiments on 650 MHz Pentium 3 based desktop computers with 128 Mb memory and on laptops with 266 MHz Pentium II processors and 96 Mb memory, all running Linux 2.4. We used Cactus/C, a C version of Cactus, and the XFree86 version 4.0.2 distribution of X including Xt, Xlib, and the Athena widget family. Cactus programs used for profiling included a H263-based video player implemented on top of a configurable transport protocol CTP [22], and SecComm, a configurable secure communication service [10]. For X-based programs, we focused on optimizing specific event responses. Programs we used for this purpose include *xterm*, a popular terminal emulator on Unix systems, and *gvim*, a graphical version of the vi text editor program.

The effect of these optimizations is more pronounced on the the slower processor based laptops. Common configurations for handheld devices (206Mhz,$\geq$64Mb RAM) are similar to those of the laptops. We expect the need for and effect of these optimizations to be even more significant on such platforms.

4.2 Cactus Programs

The video application is based on the Configurable Transport Protocol (CTP) developed using Cactus/C. The input parameters for the video player include the resolution of video and the frame rate. The experiments were carried out on two data-files of 15-16 Mb recorded at 144 $\times$ 176 resolution. Both the original and the optimized versions of the program were run 100 times each, at each of several different frame rates. Individual times per event were computed by running each program only 10 times, since each event occurs a large number of times on each run; during a run of the program about 8000 events—1000 of them asynchronous—are raised. The execution time reported, in each case, is the average of the run times so obtained. The event graph for this application contains 18 distinct events as shown in figure 5. Each event has 2–4 event handlers, with a total of 54 handlers in the system.

Figures 10 and 11 show the effects of optimizing the video player example on the laptop. The event processing times reported in figure 11 were measured when running the program with frame rate of 10 frames/second, while figure 10 shows the impact of the optimization on the total execution time of the program. Event *Adapt* has 2 handlers, while events *SegFromUser* and *Seg2Net* have 4 handlers each. In this experiment, our techniques reduce the time spent in event handlers by 73–88%. Event processing contributes to about 10–15% of the total system costs; I/O accounts for about 60% of the total execution time. There is a concomitant improvement in the overall execution time, as shown in Figure 10, ranging from 2.3% at a frame rate of 10 to about 11% for a frame rate of 25. The reason that the impact of the optimization on overall execution time becomes more pronounced as the frame rate increases is that when the frame rate is low, the CPU is idle a large part of the time. Therefore, the unoptimized program can simply use a bit more of the idle time to keep up with the required frame rate. With low frame rates, both programs can keep up with the required frame rate. However, when the frame rate increases, both programs must do more work in a time unit and the idle time decreases. When the frame rate becomes high enough, the unoptimized program runs out of extra idle time and starts falling behind the optimized program. This indicates that our optimizations are especially effective for mobile systems, such as handheld computers, that tend to have less powerful processors than, say, desktop systems.

Event	Processing Time (μsec)		Speedup
	Original	Optimized	(%)
Adapt	55	11	80.0
SegFromUser	346	41	88.2
Seg2Net	137	37	73.0

Figure 11: Event Processing Times in the video player.

SecComm is a configurable secure communication service that allows the customization of security attributes for a communication connection, including privacy, authenticity, integrity, and non-repudiation. One of the features of SecComm is its support for implementing a security property using combinations of basic security micro-protocols. We optimized a configuration of SecComm with three micro-protocols, two of which encrypt the message body (DES and a trivial XOR with a key) and the third that coordinates the execution of the other two. SecComm is a much simpler composite protocol than CTP and the video player, and the event behavior in this particular SecComm configuration turns out to be quite predictable. In particular, there is one event path on the sender and one path on the receiver. The majority of the execution time in SecComm is spent in the cryptographic encryption and decryption routines. The SecComm measurements were performed on the desktops, as follows: first a dummy message was sent to initialize the micro-protocols, after which messages were sent 100 times. This was repeated for different packet sizes, for a total of 1000 messages per packet size. The time reported in each case is the average

Size	Push time (μsec)			Pop time (μsec)		
	Orig. (T_0)	Opt. (T_1)	T_1/T_0 (%)	Orig. (T_0)	Opt. (T_1)	T_1/T_0 (%)
64	274	241	88.0	397	378	95.2
128	287	263	91.6	460	448	97.4
256	304	273	89.8	484	457	94.4
512	336	299	89.0	494	470	95.1
1024	430	373	86.7	608	570	93.8
2048	572	552	96.5	1016	893	87.9

Figure 12: Impact of optimization in SecComm

of the run times so obtained.

Figure 12 shows the amount of time spent in the *push* and *pop* portions of SecComm before and after optimization. The push portion encompasses the message processing from the time it is passed to SecComm by the application until it is passed to the UDP socket. The pop portion encompasses the message processing from the time it is received from the socket until SecComm passes it to the higher layer (the application). The push portion includes the time taken by the encryption operations, whereas the pop portion includes the decryption time. The time taken depends on the size of message packets and we present results for different packet sizes. It can be seen that the time for the push portion is reduced markedly in most cases, with improvements of up to 13.3%. The improvements in the pop portion are also noticeable, though not as high as for the push portion, typically around 5% but going as high as 12%.

An examination of the effects of our optimizations on these two programs indicates two main sources of benefits: the reduction of argument marshaling overhead when invoking event handlers; and handler merging, leading to a reduction in the number of handler invocations. The elimination of marshaling overhead seems to have the largest effect on the overall performance improvements achieved. The main effect of handler merging is to reduce the number of function calls between handlers that are executed in sequence. Merging also creates opportunities for additional code improvements due to standard compiler optimizations.

To measure the effects of our optimization on code size we counted the number of instructions in the original and optimized programs using the command `objdump -d` *program* `| wc -l`. Our optimizations produce a code size increase of 1.3% for the video player and 1.1 % for Seccomm.

4.3 X-based programs

X-based programs tend to be user driven, spending much of their time in the event loop waiting for user input. Hence our focus in the case of X-based programs is to improve the event response time, i.e., the time taken to handle an event. Common applications like *gvim* exhibit several examples of multiple handlers binding to single events and hence are are good candidates for applying such optimizations. This section is indicative of the potential of our techniques.

We evaluated our ideas on the *xterm* application provided with XFree86 and *gvim*. The effects of our optimizations on these programs while running on the laptop are shown in figure 13. These numbers were obtained by raising the events 250 times.

Popup represents the Menu Popup that is triggered of by CTRL + MOUSE_BUTTON in an *xterm* window. When handling the popup, two action handlers are triggered in sequence. The first action initializes the a menu object. This procedure is specific to the type of GUI toolkit and in our case uses the *SimpleMenu* widget in the Athena Toolkit. The next action handler is responsible for constructing and displaying the menu. This action handler in turn calls two callbacks to track mouse motion within the menu. Our optimizations merge these two action handlers as described earlier. The *Scroll* event corresponds to motion of the scrollbar in a *gvim* window. Handling this event also involves two action handlers that move the thumb[1] and update the new position. The first action handler uses the underlying framework to get the co-ordinates of the thumb. The second is responsible for displaying the new position of the thumb on the screen. Both these action handlers call callbacks tied to corresponding widgets.

Event Type	Execution Time (μsec)		T_1/T_0 (%)
	Orig. (T_0)	Opt. (T_1)	
Scroll	158	148	93.7
Popup	37	31	83.8

Figure 13: Optimization of X events

It can be seen that our optimizations reduce the cost of *Scroll* by about 6% and that of *Popup* by over 16%. Our optimizations were applied at action handler level and we can optimize one step further by opening up callbacks in the same way.

We have tried our optimizations on the Athena widget family based on Xt and Xlib, provided with XFree86. Athena toolkit is a minimal toolkit with limited scope for configurability, and therefore provided limited scope for applying our optimizations. The event model in more recent (and popular) toolkits such as Gnome GTK and KDE Qt provides functionality, e.g., *signals* and *slots*, that is very similar to the Cactus event model. Such functionality greatly increases the ease of use and development of client applications but increases the cost of event handling significantly. Application of our optimization techniques directly to these systems would reduce these costs and make event handling through signals etc. no more expensive than "ordinary" event handling while significantly enhancing ease of programming.

5. EXTENSIONS

We are working on a number of extensions to these optimization techniques. These range from simple extensions and automation of some of the steps required to extensions for dealing with the dynamic aspects of the event system and for dealing with cases where the event execution is not quite deterministic. An example of a simple extension would be to perform handler merging for all events that have more than one handler rather than only the events in frequently executed event chains. In a sense, the optimization used for X-based programs already uses this optimization.

Event-based system may change their behavior dynamically and any optimization approach must be able to handle such changes. The behavior of an event-based system can be altered simply by changing the event binding. Such a change is a challenge for an optimization system based on identifying and utilizing predictable behavior in an event system. Our current approach is based on

[1]The "thumb" here refers to a portion of the scrollbar.

detecting any change in event bindings and falling back to the original code for any events affected by the binding change. A better approach would be to construct the super handler so that it can be used even if some of the bindings change. For example, consider a predictable event sequence (A,B,C,D), which has been optimized into one super handler $(ABCD)^*$. If handlers for B change, the current approach falls back to the original code for this whole event sequence and all of the performance improvement achieved by the optimization are lost. Alternatively, we could organize $(ABCD)^*$ super handler internally so that the code corresponding to different events is partitioned as illustrated for event B in figure 14. Even if the event binding for event B changes, the optimized version can still be used for events A, C and D.

. . .
if *event binding for event B changed* {
    call(original code for event B);
} **else** {
    merged, inlined, and optimized code for event B;
}
. . .

Figure 14: Extended super handler for dynamic system

Other optimizing techniques could be added to optimize the system even further than just the predictable event paths. Once all event chains have been optimized, we will be left with a reduced graph with no event paths. This reduced graph can be further optimized by using a speculative approach. In this scheme, if node A is followed by B 90% of the time and C 10 % of the time, free cycles in A can be used to initialize the execution of B. Value based optimization, as suggested in [15] may also be extended to increase the accuracy of prediction. Another strategy would be to perform minimal processing for A and defer the bulk of handing A the next event occurs. If the next event is B, optimized code for $(A,B)^*$ can be executed. This type of deferral would particularly useful in a situation where event A is followed by B or C with equal probability. Heavier optimizations such as dominator / post-dominator analysis can be used to detect co-relations between events.

Our next step will be to optimize events activated asynchronously. As we pointed out earlier, the current optimization can only address event paths that may start with a synchronous or asynchronous activation, but all the other activations must be synchronous. Asynchronous activation, by its nature, has much looser semantics than a synchronous activation; the handlers bound to this event must be executed some time, preferably soon, after the event has been activated. Although these semantics will make it difficult to create an optimized program that has identical behavior to an unoptimized one, it should be possible to generate one that executes the events and handlers in order acceptable for the semantics of the event operations. In particular, if an asynchronous activation of an event A is always eventually followed by the activation of event B, it may be semantically correct to merge the handlers of A to the end of the handlers for B. We will explore which program transformations such as this preserve program correctness. We also plan to extend the family of compiler optimizations to include techniques such as register allocation etc which are currently effected through standard compilers. Finally, we plan to extend this work to include dynamic profiling tool along with optimization modules for the compiler. This will be used to apply our techniques to other event based systems like system call and interrupt behavior of operating systems.

6. RELATED WORK

We are not aware of a great deal of work aimed specifically at compilation oriented optimization of event-based systems. Chambers *et al.* discuss the use of dynamic compilation to optimize event dispatching in the SPIN operating system [4]. Unlike our work, which uses profile-based static optimizations, the work of Chambers *et al.* relies on optimizations that are carried out during execution. As with other dynamic optimization systems (e.g., Dynamo [1], Tempo [6]), the benefits of dynamic optimization have to be balanced against the overheads associated with runtime monitoring, optimization, and code generation. Because of this, dynamic code optimization systems generally rely on lightweight optimizations. By contrast, the optimizations we carry out are fairly heavy-weight, with correspondingly high payoff.

Events, and their variants, have traditionally been used for interaction between different layers of systems software. For example, Ensemble [8] uses events for communication between layers in a protocol stack. Ensemble protocol stacks are typically relatively deep since they consist of rather small modules, each of which implements a specific function. Furthermore, Ensemble is implemented using ML, a functional programming language. Ensemble focuses optimizations on the common sequence of operations (the "normal" case) that occur in a protocol stack. Each such sequence, denoted *event trace*, is triggered by an event such as receipt of a message. The common operations must be annotated by the protocol designer. The code in an event trace is optimized into one *trace handler* using techniques ranging from eliminating intermediate events to inlining and traditional compiler optimization. Each event trace has a *trace condition* that must be true for the trace handler to be executed. The trace condition must have predicates from each protocol layer and these predicates must be provided by the protocol designer. In comparison, our approach does not require the protocol designer to provide any annotations or predicates. Furthermore, out focus is on providing generally applicable optimizations techniques for event-based systems rather than for any one specific system.

Synthesis [20] and Synthetix [19] were efforts at specializing operating systems through manual specialization. The primary disadvantage is the loss of portability and maintainability. Automated specialization is gaining importance especially in areas related to byte code interpretation. Most of these approaches rely on partial evaluation based on knowing the input values.

Event-based architectures are gaining popularity in designing complex software systems. Event-based web servers of the SEDA project [21] outperform their thread-based counterparts in terms of responsiveness and scalability. This approach partitions a system into many logically independent *stages*, where the different stages communicate using events. Each stage is serviced by its own thread pool. Conceptually this is the opposite of our approach that attempts to reduce the number of events in the system. The primary difference is the targeted system architecture. While our work targets small single processor, possibly single-threaded, systems as expected on a mobile device, SEDA targets high-end server architectures with several SMPs. SEDA gains its performance improvement from maximizing parallelism by exploiting free cycles available on the SMPs and hence they gain by releasing the processor as soon as possible. Our approach gains from traditionally optimizing program behavior, e.g., elimination of procedure calls and other redundant code, and attempting to complete as much work as possible before yielding the processor. Our performance results in table 10 partially confirm the findings in [21]. When the frame rate is very low, our optimization actually slightly increases the execution time because locks are held longer by super handlers resulting in starving other threads even though the CPU utilization is low. As the frame rate and CPU utilization increase, the effect of our optimizations gets more pronounced until we reach a limit where system throughput becomes the limiting factor.

Finally, ILP (Integrated Layer Processing) [5] can be viewed as related work. ILP integrates data manipulation across protocol layers to minimize memory references by merging the message data manipulation done on different layers into one loop where each data item is accessed only once. This technique can be used, for example, to merge encryption, checksum computation, compression, and presentation formatting into one loop. Our optimization does not specifically attempt to reduce memory references related to accessing the message data, so ILP could in principle added as an additional optimization technique to our approach. Note that profiling the memory references in a super handler the same way we profile events and handlers, could automate the process of finding candidate code for ILP optimization.

7. CONCLUSION

Event-based programs are used in a variety of domains due to their flexibility. However, event-based program execution has a considerable performance overhead. Due to the dynamic nature and unpredictability of events—which event occurs and when—and event bindings—which handlers are bound to an event when it occurs—compiler optimization techniques have not traditionally been used in this domain. However, in this paper, we have determined that many event-based programs have considerable amount of predictability that can, indeed, be utilized to perform powerful compiler optimizations. We have developed a novel multi-resolution profiling technique to identify predictable program sequences and thus, targets for the optimization. Through the application of simple compiler layout and dataflow optimizations, we have achieved up to 80% reduction of event handling time, resulting in overall program performance improvements of up to 12%. We have successfully applied our general optimization technique to two event-based systems, namely Cactus and X. We believe this work will the first in a series of tackling performance issues in event-driven programs and system software.

8. REFERENCES

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[3] N. Bhatti, M. Hiltunen, R. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, Nov 1998.

[4] C. Chambers, S. Eggers, J. Auslander, M. Philipose, M. Mock, and P. Pardyak. Automatic dynamic compilation support for event dispatching in extensible systems. In *Proceedings of the 1996 Workshop on Compiler Support for Systems Software (WCSSS-96)*, Feb 1996.

[5] D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, September 1990. ACM.

[6] C. Consel, L. Hornof, J. Lawall, R. Marlet, G. Muller, J. Noy, S. Thibault, and E.-N. Volanschi. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation (SOPE '98)*, 30(3), Sep 1998.

[7] Microsoft Corporation. *Microsoft Visual Basic 6.0 Programmer's Guide*. Microsoft Press, Aug 1998.

[8] M. Hayden. The Ensemble system. Technical Report TR98-1662, Department of Computer Science, Cornell University, Jan 1998.

[9] M. Hiltunen, R. Schlichting, X. Han, M. Cardozo, and R. Das. Real-time dependable channels: Customizing QoS attributes for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):600–612, Jun 1999.

[10] M. Hiltunen, R. Schlichting, and C. Ugarte. Enhancing survivability of security services using redundancy. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2001)*, pages 173–182, Gothenburg, Sweden, Jul 2001.

[11] M. Hiltunen, R. Schlichting, and G. Wong. Cactus system software release. http://www.cs.arizona.edu/cactus/software.html, Dec 2000.

[12] J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 259–269, Atlanta, Georgia, May 1–4, 1999.

[13] R. Marlet, S. Thibault, and C. Consel. Efficient implementations of software architectures via partial evaluation. *Journal of Automated Software Engineering (J.ASE)*, 6(4):411–440, Oct 1999.

[14] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 707–710, Phoenix, AZ, Apr 2001.

[15] R. Muth, S. A. Watterson, and S. K. Debray. Code specialization based on value profiles. In *Static Analysis Symposium*, pages 340–359, 2000.

[16] A. Nye and T. O'Reilly. *X Toolkit Intrinsics Programming Manual*. O'Reilly and Associates, 1992.

[17] Object Management Group. *Event Service Specification (Version 1.1)*, March 2001.

[18] J. Ousterhout. Why threads are a bad idea (for most purposes). In *1996 USENIX Technical Conference*, Jan 1996. Invited Talk.

[19] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic incremental specialization: Streamlining a commercial operating syste. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 314–324, Copper Mountain, CO, Dec 1995.

[20] C. Pu, H. Massalin, and J. Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[21] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18)*, Banff, Canada, Oct 2001.

[22] G. Wong, M. Hiltunen, and R. Schlichting. A configurable and extensible transport protocol. In *Proceedings of the 20th Annual Conference of IEEE Communications and Computer Societies (INFOCOM 2001)*, pages 319–328, Anchorage, Alaska, Apr 2001.

[23] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, pages 180–190, June 20–22 2001.