Profile-Directed Optimization of Event-Based Programs

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

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

Abstract

Events are used as a fundamental abstraction in programs ranging from graphical user interfaces (GUIs) to systems for building customized network protocols. While providing a flexible structuring and execution paradigm, events have the potentially serious drawback of extra execution overhead due to the indirection between modules that raise events and those that handle them. This paper describes an approach to addressing this issue using static optimization techniques. This approach, which exploits the underlying predictability often exhibited by event-based programs, is based on first profiling the program to identify commonly occurring event sequences. A variety of techniques that use the resulting profile information are then applied to the program to reduce the overheads associated with such mechanisms as indirect function calls and argument marshaling. In addition to describing the overall approach, experimental results are given that demonstrate the effectiveness of the techniques. These results are from event-based programs written for X Windows, a system for building GUIs, and Cactus, a system for constructing highly configurable distributed services and network protocols.

1 Introduction

Events are increasingly used as a fundamental abstraction for writing programs in a variety of contexts. They are used to structure user interaction code in GUI systems [10, 19], form the basis for configurability in systems to build customized distributed services and network protocols [5, 12, 18], are the paradigm used for asynchronous notification in distributed object systems [20], and are advocated as an alternative to threads in web servers and other types of system code [21, 27]. 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 multifaceted. 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 disadvantage of extra execution overhead due to the indirection between modules that raise and handle events [7, 16]. 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 graph* that is conceptually akin to the call graph of the program. This graph and an analogous *handler graph* are used to identify commonly encountered events and event sequences, as well as the collection of handlers associated with each event and the order in which they are executed. This information is then used to optimize event execution by, for example, merging handlers and chaining events. Although these optimizations are currently implemented by manual source code rewriting, automation is straightforward.

The techniques presented 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 [2] can be used in principle, but, in an effort to keep runtime overheads low, they focus primarily on lightweight optimizations such as improving locality and instruction-cache usage. 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 especially useful for resource-constrained devices, where any reduction in overhead is valuable.

The remainder of the paper is organized as follows. Section 2 describes a general model for event-based programs. This is followed by a description of our approach to profiling such programs in section 3 and techniques for optimizing them in section 4. Section 5 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 [13, 15]. The third is a client side tool that uses X Windows, a popular system for building GUIs [19]. This is followed by a discussion of other systems for which this approach might be useful and related optimization work in section 6. Finally, section 7 offers conclusions.

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 Cactus and the X Windows system systems map into the model. Section 6 outlines how other event-based systems map into this model and thus, how the same optimization techniques could be applied in these systems.

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. Events may be classified as *external* or *internal* depending on the nature of the occurrence. Mouse motion, button click, and key press are examples of external events in a user interface context, while receiving a packet from the network and system interrupt are examples in a systems context. Internal events in contrast are generated and processed within the program. For example, passing a message to a higher layer in a network stack is an example of an internal event. 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. A handler specifies the reaction of a program to an event. Specifically, a handler is a section of code, usually implemented as a procedure, that specifies the actions to be performed when an event occurs. Event handlers have a simple basic block structure with a single entry and exit points. Typically, handlers have at least one parameter, the event that was raised; other parameters may be passed explicitly through variable argument lists or through shared data structures or implicitly within the event data structure. The decoupling provided by the event mechanism allows more than one handler to be associated with the same event, each developed independently from other handlers in the program.

Bindings. Bindings determine which handlers are executed when a given event occurs. Most systems allow multiple handlers to be bound to a single event and a handler to be bound to more than one event. If no handlers are bound to an event when it is raised, the occurrence is treated as a no-op. The order of executing multiple handlers bound to an event may be important depending on the semantics of the program.

Although the binding of handlers to events may be predefined and fixed, often, the binding is established at runtime

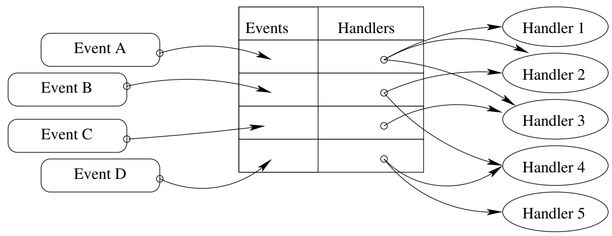

Figure 1: Event bindings

using some type of *bind* operation. In its simplest form, the bind operation takes two parameters, the handler and the corresponding event, and adds the handler of a list of handlers associated with the event. If a simple FIFO execution order is not sufficient, the bind operation may take a third parameter that specifies in some way (e.g., a priority) the order in which handlers should be executed when an event occurs.. Bindings may be static, i.e., remain the same throughout the execution of the program, or dynamic, i.e., change at runtime. The set of bindings valid at a given point in time during program execution is called the *event configuration* of the program. The use of runtime binding means that the program's behavior can be changed dynamically by changing this configuration from within the program.

Figure 1 illustrates 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

An event may occur because the program receives some external stimulus (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.

Events are usually activated via some type of *raise* operation, and results in all of the handlers bound to that event being executed. There are two types of event activation, *synchronous* and *asynchronous*, which differ in the way the handlers for the activated event are executed. Intuitively, with synchronous activation the handlers are executed to completion before execution of the code doing the raise continues, while with asynchronous activation further execution of the code may continue concurrently with execution of the handlers. More precisely, the operational behavior can be described as follows:

- The synchronous activation of an event e from a program point p causes control to be transferred to the handlers of e. These are then executed in sequence one after another, until they have all completed execution, after which control is returned to the program point immediately after p.
- The asynchronous activation of an event e from a program point p causes execution of the handlers to be scheduled, but not necessarily immediately. Consequently, execution after program point p may continue before the handlers are executed.

Note that this notion of (a)synchrony describes the way in which an event is activated, not the event itself; in other words, it is possible for an event to be activated synchronously at one point and asynchronously at another.

This paper focuses primarily on synchronous event activations. For such activations, we assume that all the handlers executed as a result of an event occurrence—including nested synchronous activations—are executed as an atomic unit with respect to other events in the system. This, combined with the operational behavior described above, implies that for a set of synchronous event activations, the lifetimes of the activations are either nested or disjoint, i.e., follow a LIFO discipline very similar to that of subroutine activations in procedural programming languages. This insight makes it possible to adapt many interprocedural optimizations, such as subroutine inlining, to synchronously activated events.

The different types of event activation have specific uses in event-based systems. For internal events, synchronous activation can be used when the event activator needs to know when the processing of, for example, a message has completed before continuing its own processing. Similarly, for external events, synchronous activation can be used 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. Although both forms of activation provide a high degree of flexibility and configurability, synchronous activation is easier to optimize and provides low handling latency, whereas asynchronous activation is preferred when scalability is a concern.

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 the event at that time results in a correct transformation. Similarly, it is easy to see that sequences of nested synchronous activations can be readily optimized. The specific optimization techniques and their limitations are discussed below in section 4.

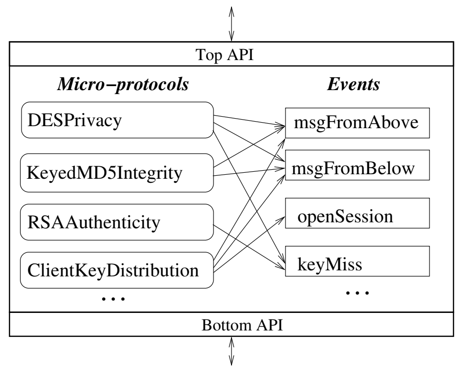

Figure 2: Cactus composite protocol

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 [13]. 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 a 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 5 gives concrete 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 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.

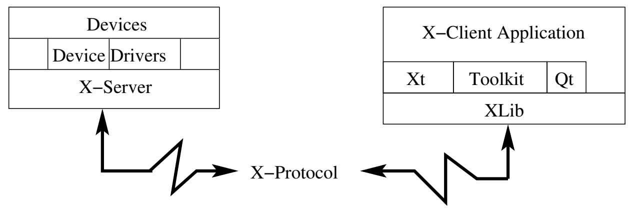

Figure 3: Architecture of X Window systems

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 such as 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" [19]. 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 that are specified at bind time. Events are also used for communication between widgets. Events can arrive in any order and are queued by the X client. Event activation in X is similar to synchronous activation in the general model.

The X architecture has three mechanisms for handling events: *event handlers*, *callback functions*, and *action procedures*. All these map to handlers in the general model and are used to specify different granularities of control. Event handlers, the most primitive, are simply procedures bound to event names. Callback functions and action procedures are more commonly used high-level abstractions. One difference between the three mechanisms relates to scope—actions have global scope in an X client, while the scope of event handlers and callbacks is restricted to the widget in which they are defined. Another difference is their execution semantics. An event handler can be bound to multiple events in such a way that it is executed when *any* of the associated events occur. A callback function, on the other hand, is bound to a specific callback name, and all functions bound to a name are executed when the corresponding callback is issued. 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.

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 a given condition occurs. For all these handler types, X provides operations for registering the handlers and activating them.

3 Profiling Events and Event Handlers

Compiler optimizations are based on being able to statically predict aspects of a program's runtime behavior using either invariants that always hold at runtime (i.e., based on dataflow analysis) or assertions that are likely to hold (i.e., based on execution profiling). Compiler optimization has traditionally not been applied for event-based systems because of their perceived unpredictable runtime behavior due to the uncertainties associated with the behavior of their external environment, e.g., the user's actions. However, we have found 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 occur in all (or most) system executions, that is, we can often identify significant correlations between different events, of the form *"Event e_2 always follows Event e_1."* At the handler level, there is often more than one handler bound to a specific event, and all these handlers are executed in sequence each time the event occurs. Note, however, that the event bindings are typically only created at runtime and may change during program execution.

We identify predictable aspects of a program execution, and thus, static optimization opportunities, by profiling both event and handler execution. This is done in two steps. First, the program is profiled to identify commonly encountered sequences of events. Second, this information is used to target specific handlers in the program for similar profiling to identify predictable sequences of handlers that can be optimized. At present, programs are instrumented by hand, but this can easily be automated using well-understood techniques [3]. The remainder of this section elaborates on these profiling steps.

Event execution is profiled by instrumenting the event system to create an *event trace* each time a program is executed. This trace consists of a sequence of entries, where each entry corresponds to an occurrence of the *raise* or *bind* operations. An entry contains relevant information about the operation, including for *raise* the event name and whether it has been activated synchronously or asynchronously, and for *bind* the event name and handler address. Figure 4 gives a portion of an example event trace generated from the execution of a Cactus-based video player application implemented on top of CTP (a configurable transport protocol) [28]. This application is described in more detail in section 5.2.

The actual profiling of a program is performed by executing the instrumented version repeatedly with different inputs to generate a collection of event traces that characterize the event behavior of the program. This collection of traces is then used to construct an *event graph* that summarizes the event sequences in the traces and forms the basis for all further analysis. The algorithm used to generate the event graph from the traces is presented in figure 5. Each node in an event graph represents a unique event. There is an edge from node e_1 to node e_2 in the graph if event e_1 is followed immediately by event e_2 at some point in one of the traces. Each edge (e_1, e_2) has an associated weight indicating how

```
    ...
RaiseEvent (pev= 0x832f9f0 MsgFromUser, em= SYNC, Delay= 0)
RaiseEvent (pev= 0x832fab8 SegmentFromUser, em= SYNC, Delay= 0)
RaiseEvent (pev= 0x832fb08 SegmentToNet, em= SYNC, Delay= 0)
RaiseEvent (pev= 0x832fdf0 Controller Clock, em= ASYNC, Delay= 33)
RaiseEvent (pev= 0x832feb8 ControllerClock, em= SYNC, Delay= 0)
RaiseEvent (pev= 0x8332570 ControllerClock, em= SYNC, Delay= 0)
RaiseEvent (pev= 0x83324f8 Adapt, em= SYNC, Delay= 0)
RaiseEvent (pev= 0x832fe68 ControllerFiring, em= SYNC, Delay= 0)
RaiseEvent (pev= 0x832fe90 ControllerFired, em= SYNC, Delay= 0)
RaiseEvent (pev= 0x83324d0 MsgFromUser, em= SYNC, Delay= 0)
RaiseEvent (pev= 0x832f9f0 MsgFromUser, em= SYNC, Delay= 0)
    ...
```

Figure 4: A segment of an event trace.

```
EventGraph = ∅;
for each eventTrace in collection{
    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 5: *GraphBuilder* algorithm.

many times the sequence $\langle e_1, e_2 \rangle$ appears in all the traces. Apart from the number of occurrences of events, the type of each event activation is also crucial for correctly describing the behavior of a program. If an event e_1 is followed immediately by a synchronously raised event e_2 in the event graph, we can infer that execution of e_2 sequentially follows that of e_1. This is not true of asynchronous activation, however: if e_2 is raised asynchronously, then the fact that it follows e_1 in the event graph need not indicate causality, and we cannot conclude that e_1 had any role in raising e_2. For example, e_2 may be the result of a timeout from an earlier event completely unrelated to e_1. Figure 6 shows the event graph for the Cactus-based video player application.

In general, a program may contain many events and associated handlers, and the corresponding event graph can be quite large. Pragmatically, however, the portion of the graph representing occurrences that are common enough to be interesting from the perspective of code optimization is usually quite small. To identify the "hot," or "commonly occurring," components of the event graph, we start with a (user-defined) fraction θ ($0.0 < \theta \leq 1.0$) that specifies what fraction of the total number of event raises in the event graph should be accounted for by the portion of the event

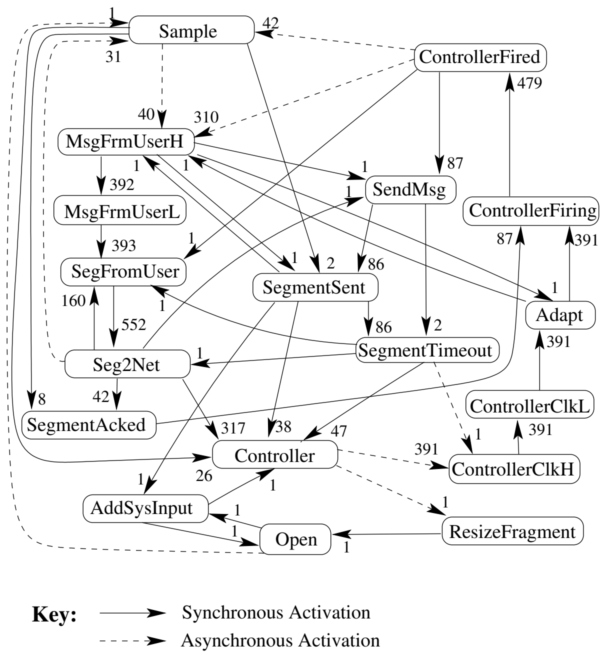

Figure 6: Event graph generated from video player

graph considered to be "hot." For example, $\theta = 0.8$ would mean that the hot components should account for at least 80% of the event raises in the event graph. More formally, given a value of θ for an event graph with set of edges E, we consider all edges $e \in E$ in decreasing order of weight, and determine the largest weight η such that

$$\sum_{e \in E:\, weight(e) \,\geq\, \eta} weight(e) \quad \geq \quad \theta \cdot \sum_{e \in E} weight(e)$$

An edge is considered to be "hot" only if its weight is at least η. We refer to the subgraph of the event graph induced by the set of hot edges so identified as the *reduced event graph*. Figure 7 shows the reduced event graph for the graph of Figure 6, corresponding to a value of $\theta = 0.8$, which yields an edge weight threshold of $\eta = 310$: light edges and vertices represent cold components of the graph, while dark edges and vertices correspond to hot ones.

The next step is to perform handler level profiling to identify predictable sequences of handler activations. This profiling is needed for three reasons:

- **Handler Bindings.** Because of the decoupling between events and their handlers, knowing the events that occur does not by itself tell us about the order in which handlers are activated.

- **Execution Order.** An event may have multiple handlers that are executed each time the event occurs. Determining the order in which these handlers are executed is fundamental to applying optimization techniques to the sequence, but cannot be determined from the event graph.

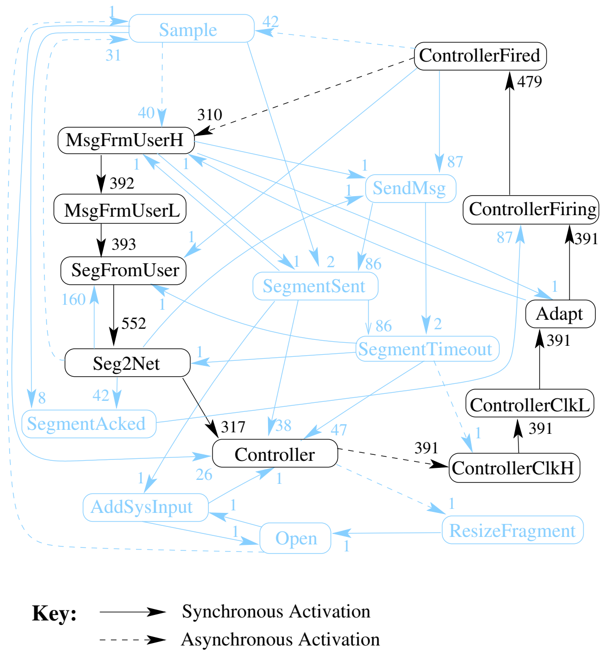

Figure 7: Reduced event graph

- **Embedded Activations.** The event graph does not capture the execution order of handlers associated with synchronous *embedded activations*, i.e., when an event e_2 is synchronously activated from within a handler for another event e_1. In this case, the relative ordering of e_1's and e_2's handlers depends on which handler raises e_2, something that cannot be determined from the event graph. A concrete example is discussed further below in conjunction with figure 9.

Handler-level profiling is done by instrumenting handlers using an approach similar to that described above for event profiling. The particular handlers to be instrumented are those associated with events in the reduced event graph, which identifies the most commonly occurring event execution sequences. These handlers are determined from the binding information collected during the event profiling phase. An entry in the *handler trace* is generated whenever a handler is dequeued for execution, and includes the address of the handler and the event with which it is associated.

The trace information is used to construct a *handler graph* in much the same way as done for events. In this graph, nodes represent handlers and edges denote the order of execution. Edges again have counts that indicate the frequency with which that sequence is encountered during the profiled executions.

4 Event Handler Optimizations

Once the most frequent event and handler sequences have been identified, the optimizations can be performed. The goal of the optimizations is to eliminate:

- Marshalling overheads for event activations.
- Indirect function call and variable argument passing costs.
- State maintenance (synchronization and locking) costs for global variables.
- Redundant initializations and code fragments for events with multiple handlers.

The elimination of indirect function calls also increases the potential for value-based optimizations such as constant propagation, and makes it possible to inline the code for raising common events. All these optimizations can be classified broadly as either graph or compiler optimizations. In the following, we first discuss correctness issues underlying our optimizations, then describe each type of optimization in turn.

4.1 Correctness Issues

Profile-based optimizations, by their very nature, are based on information about a program's behavior on some set of sample inputs. While the goal of the profiling is to capture typical behavior, there is no guarantee that every execution will follow this pattern. For our approach, the key issue is whether the event configuration—i.e., the collection of event-handler bindings—at a given point during execution matches the configuration that was optimized. It might not, for example, if a new handler is added for some event at run time in a way that differs from the profiled runs.

Since the optimized program must operate in the same way as the corresponding unoptimized version even in these cases, there must be a mechanism for ensuring that the optimized code is only used when the unoptimized program would have taken the corresponding sequence of handlers. To deal with this possibility, we associate with each event e an "optimize bit" $e.opt$ that indicates whether the current handler binding for event e is the same as the handler binding that was optimized. Any event e whose optimization-time bindings differ from its current execution-time bindings has the corresponding $e.opt$ bit set to false. This bit is updated every time the event handler bindings change due to the runtime addition or removal of handlers.

The optimize bit $e.opt$ for an event e is used to ensure that the optimized handler code for e is executed only if the set of handlers bound to e at runtime is the same as those with with it was profiled and optimized. This is done, as discussed below, by introducing checks to test $e.opt$ before executing the optimized code.

4.2 Graph Optimizations

Graph optimizations reduce the costs associated with interactions between events and handlers in the program by reducing the number of handler activations along common event execution paths. This is done by *handler merging*, which reduces the number of handlers for an event by merging handlers to create *super-handlers*, and by *raise elimination*, which eliminates synchronous raise operations, possibly leading to handler merging across multiple events.

Handler Merging. In the 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, the handler graph identifies 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. In the handler graph, this corresponds 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.

Thus, suppose that the following set of handlers $\{h_1, h_2, h_3\}$ is bound to an event e:

h_1(*argc*, *argv*)
{
    $Body_1$;
}
	h_2(*argc*, *argv*)
	{
	    $Body_2$;
	}
		h_3(*argc*, *argv*)
		{
		    $Body_3$;
		}

Straightforward merging of these handlers would yield the code

h_{123}(*argc*, *argv*)
{
    $Body_1$;
    $Body_2$;
    $Body_3$;
}

The resultant code h_{123} is the super-handler for the event. Notice, however, that in order to maintain correctness, this code should be executed only if, at runtime, the set of handlers bound to the event e is $\{h_1, h_2, h_3\}$; any runtime change to the event's handler bindings should result in execution proceeding according to the new configuration. To realize these semantics, the raise operation is guarded to check for changes in the event's bindings. If the current binding is the same as that which was optimized, the super-handler is invoked; otherwise, the new bindings are obtained by referencing the registry and executed, corresponding to the original raise mechanism. Conceptually, the optimization can be thought of as introducing a new event s_e and binding the super-handler for event e to it. The raise operation for event e can now be replaced with a raise to either the new event s_e of the optimized bindings hold or else to the original raise of event e.

This optimization can be applied to both synchronously and asynchronously activated events. In the case of synchronously activated events, further optimization is possible—the cost of the indirect function call incurred with *raise* can be eliminated by replacing *raise* with a direct call to the super-handler. This in turn opens up the possibility for inlining the function call into the call site resulting in code as shown below:

$$
\begin{array}{lclcl}
\begin{array}{l} \ldots \\ \ldots \\ \mathsf{raise}(e,\ldots); \\ \ldots \\ \ldots \end{array}
& \xrightarrow[\textit{merging}]{\textit{handler}} &
\begin{array}{l} \ldots \\ \textbf{if}\ (e.opt)\{ \\ \quad \mathsf{raise}(s\_e,\ldots); \\ \} \\ \textbf{else}\ \{ \\ \quad \text{raise(e,}\ldots\text{);} \\ \} \\ \ldots \end{array}
& \xrightarrow[\textit{elimination}]{\textit{raise}} &
\begin{array}{l} \ldots \\ \textbf{if}\ (e.opt)\{ \\ \quad h_{123}(argc, argv) \\ \} \\ \textbf{else}\ \{ \\ \quad \mathsf{raise}(e,\ldots); \\ \} \\ \ldots \end{array}
\end{array}
$$

This transformation is the topic of the next discussion.

Raise Elimination. As discussed in section 2.2, the operational behavior of synchronous event activations is in many ways analogous to that of subroutine calls in procedural programming languages. There are two main differences between subroutine calls and event activations:

(i) A subroutine call results in the execution of a single block of code, namely, the body of the called subroutine. In contrast, a synchronous event activation results in the execution of a sequence of different handlers for that event.

(ii) The code to be executed upon a subroutine call is fixed and does not change at runtime. However, the set of handlers bound to an event can change dynamically, and the process of determining which handlers to execute when an event is activated must account for this possibility.

Once handler merging has been carried out as described above, however, the first of these differences is eliminated and the second is finessed away. Since handler merging results in a single block of code consisting of the merged bodies of the event's handlers, the activation of an event now results in the execution of a single block of code—the super-handler. Since the optimized code of the super-handler for an event e is guarded by a test of the optimize bit *e.opt* as discussed in section 4.1, the possibility of runtime changes to the handler bindings for e are explicitly dealt with. As a result, we can replace a synchronous raise operation of an event e by (an appropriate instance of) the body of e's super-handler as described above.

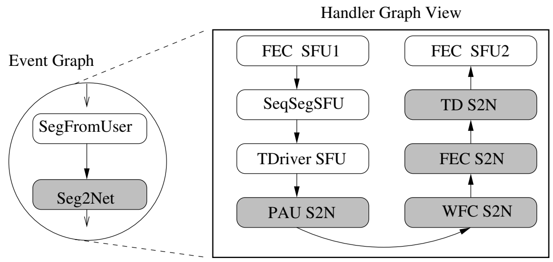

Figure 8: Raise elimination

This transformation can also be applied to synchronous raise operations within event handlers, resulting in the code for the super-handler for an event e being embedded in the super-handler for a parent event e'. Thus, consider an event e_1 whose super-handler after handler merging is h_1, as shown below:

h_1(*argc*, *argv*)
{
    S$_1$;
    raise(e_2, SYNC, *args*);
    S$_2$;
}

Suppose that the super-handler for e_2 is h_2. Then, raise elimination applied to the synchronous raise operation of e_2 in the body of the handler results in the following optimized code:

h_1(*argc*, *argv*)
{
    S$_1$;
    if (e_2.*opt*){
        $h_2(\ldots)$;   /* *direct call to superhandler for* e_2 */
    }
    else {
        raise($e_2, \ldots$)
    }
    S$_2$;
}

Notice that the synchronous raise of e_2 in the body of the handler h_1 has been replaced. If the bindings for e_2 have not changed—verified by checking the bit e_2.*opt*—the super-handler h_2 for event e_2 can be called directly, otherwise the original raise call is used to activate the event.

An example of this is found in the video player example shown in figure 6 and is illustrated in figure 8. The focus is on two distinct events from figure 6, *SegFromUser* and *Seg2Net*; the former is shown unshaded, the latter is shaded gray. The relevant portion of the corresponding handler graph is shown at the right of figure 8, with handlers executed

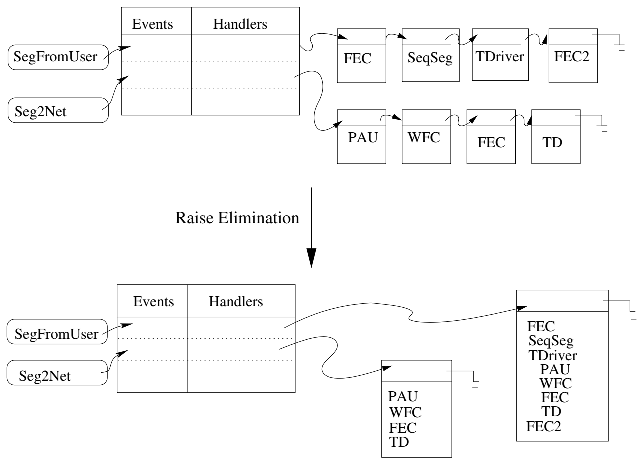

Figure 9: Effect of raise elimination in an event handler

based on the actions of *SegFromUser* shown unshaded and those executed based on *Seg2Net* shown shaded. The synchronous raise of the *Seg2Net* event by the handlers of the *SegFromUser* event can be replaced by the code for the superhandler for *Seg2Net*. Figure 9 illustrates the result. *SegFromUser* and *Seg2Net* each have four handlers in this case: *FEC-SFU1*, *SeqSegSFU*, *TDriver-SFU*, and *FEC-SFU2* for *SegFromUser*, and *PAU-S2N*, *WFC-S2N*, *FEC-S2N*, and *TD-S2N* for *Seg2Net*. *TDriver-SFU* synchronously raises the event *Seg2Net*, which causes execution of its handlers. After completion, control returns to handling *SegFromUser* and causes the last handler *FEC-SFU2* to be executed.

It is important to note that inter-event handler merging using the raise elimination transformation is only performed for synchronous raises. This is necessary to preserve the observable behavior of the system with respect to timing semantics. For example, suppose that an event e_1 signifying "data transfer initiated" is always followed by an event e_2 signifying "data transfer completed," but that e_2 occurs (at least) some fixed time after e_1. In this case, e_2 is raised asynchronously, and this raise operation cannot be eliminated. Nonetheless, even in this case it may still be possible to eliminate some of the overhead associated with the raise operation based on knowledge about the context in which later events can be raised. For instance, when event e_2 in this example is raised, the event e_1 must have been raised and handled previously.

These optimizations result in an increase in code size, since the program must now contain both the super-handlers and the original unoptimized handlers. However, our experiments indicate that this increase is small, typically under 1.5% (see figure 13 in section 5).

4.3 Compiler Optimizations

The super-handlers resulting from graph optimization have the effect of bringing together code that was scattered across the program over a number of different handler routines prior to optimization. As a result, they become amenable to further improvement using standard compiler optimization techniques. Here, we describe some optimizations that tend to be especially useful in this context. The primary motivation in choosing these techniques is their relevance to the expected gains based on the nature of code in event handlers.

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. Event-based systems typically do not regulate the number and types of parameters for a handler, which leads to the use of variable argument lists. Inlining handler code allows the elimination of 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.

Furthermore, event handlers are often shared across different events, with conditional statements used to determine the steps used to handle each event. 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. Structuring programs using events promotes the design of event handlers that are largely independent from one another. Because of this, they are usually written without making assumptions about actions that may be carried out by other handlers. This can lead to redundant code in the program. 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 program state: since the program state may be changed by handlers, in a multithreaded context handlers typically start by updating their state from the global state, and commit a new state at the end. In an event chain, this can cause redundant updates. Such redundant operations can be identified in the super-handler code and targeted for elimination.

Code Layout. Once the code for different handlers has been brought together, the resulting code can be laid out in memory in a way that improves the instruction cache utilization of the program.

The use of a dynamic optimization system and techniques such as speculation, partial evaluation, and dominator analysis to predict event sequences can also be considered in this context.

5 Experimental Results

5.1 Overview

We evaluated our ideas using Cactus/C, the C version of the Cactus system [13], and the XFree86 version 4.0.2 distribution of X including Xt, Xlib, and the Athena widget family. Experiments were run on 650 MHz Pentium 3 desktop machines with 128 MB memory and on laptops with 266 MHz Pentium II processors and 96 MB memory, all running Linux 2.4. The Cactus programs used for profiling included a H263-based video player implemented on top of a configurable transport protocol called CTP [28], and SecComm, a configurable secure communication service [14]. For X-based programs, we focused on optimizing specific event responses. Programs used for this purpose included *xterm*, a popular terminal emulator on Unix systems, and *gvim*, a graphical version of the vi text editor program.

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

5.2 Cactus Programs

The video application displays video stored as a file on a local disk. The input parameters for the video application include the resolution of the video and the frame rate. The experiments were carried out on two data files of 15-16 Mb recorded at 144×176 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 6. Each event has 2–4 event handlers, with a total of 54 handlers in the program.

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 accounts for about 10–15% of the total time, while I/O accounts for

Frame rate	Execution Time (sec)		Impr.	Handler Time (sec)		Impr.
	Orig.	Opt.	%	Orig.	Opt.	%
10	43.1	41.9	2.8	2.3	0.9	60.9
15	30.9	30.3	2.0	1.6	0.6	62.5
20	24.5	22.1	9.8	1.5	0.5	66.7
25	23.9	21.3	10.9	1.5	0.5	66.7

Key: Orig: Original program; Opt: Optimized program; Impr: Improvement

Figure 10: Video player optimization results.

Event	Processing Time (μsec)		Improvement
	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.

about 60%. 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. As a result, the unoptimized program can simply use a bit more of the idle time to 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 PDAs that tend to have less powerful processors than desktop systems.

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 chain on the sender and one chain 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 a message was 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 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.

Size	Push time (μsec) Orig.	Opt.	Impr. %	Pop time (μsec) Orig.	Opt.	Impr. %
64	274	241	12.0	397	378	4.8
128	287	263	8.4	460	448	2.6
256	304	273	10.2	484	457	5.6
512	336	299	11.0	494	470	4.9
1024	430	373	13.3	608	570	6.2
2048	572	552	3.5	1016	893	12.1

Figure 12: Impact of optimization in SecComm

Program	Original	Optimized	Increase (%)
Video	144034	145777	1.2
SecComm	159522	159682	0.1

Figure 13: Effect of optimization on program code size

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, which is reflected in the results. 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 although 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 that leads 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.

Code in event handlers is usually a small fraction of the total program size. 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`. The corresponding increase in code size is shown in figure 13.

5.3 X Clients

X clients 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 these 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 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 running on the laptop are shown in figure 14. These numbers were obtained by raising the events 250 times.

Popup represents the Menu Popup that is triggered by CTRL + MOUSE_BUTTON in an *xterm* window. When handling this event, two action handlers are triggered in sequence. The first initializes the 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 invokes two callbacks to track mouse motion within the menu. Our optimizations merge these two action handlers as described above. 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 invoke callbacks tied to corresponding widgets.

It can be seen that the optimizations reduce the cost of *Scroll* by about 6% and that of *Popup* by over 16%. These techniques were applied here at the level of action handlers, although it would be possible to optimize one step further by opening up callbacks in the same way.

Event Type	Execution Time (μsec)		Improvement
	Orig.	Opt.	%
Scroll	158	148	6.3
Popup	37	31	16.2

Figure 14: Optimization of X events

6 Discussion

This section first discusses other event-based systems that could potentially be optimized using the approach described in this paper, and then gives an overview of other work directly related to optimizations of this type.

6.1 Event-based Systems

Events have been used as fundamental abstractions in both systems and programming languages. For systems, events are often used to provide an efficient, flexible, and scalable communication paradigm, while for programming languages and toolkits, events provide for richer execution semantics. We now elaborate on each of these areas in turn.

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

Systems. Events have traditionally been used to provide a high degree of extensibility and configurability in systems such as Cactus, its predecessor Coyote [6, 5], Legion [11], Ensemble [12] and SPIN [4]. The event model in Coyote is very similar to Cactus and the general event model described in this paper, and thus, the same optimizations could be applied directly. The event model in Legion, a large-scale metacomputing platform, is similar to Coyote and the basic components of their event model [26] map directly into the general model as well.

Events and their variants have also been used as a mechanism for implementing interactions between software layers in hierarchically-structured systems. Ensemble [12], a group communication toolkit, uses events implemented as fixed size ML records to structure directed communication between layers in a protocol stack. An Ensemble protocol stack can be visualized as a set of layers, where each layer contains a collection of event handlers servicing two directed event queues (upwards and downwards). Each layer contains about 10 event handlers registered to a well-defined interface to handle interactions with lower and higher layers.

The use of event-based semantics is not new in an operating systems context. Ousterhout, for example, has suggested the use of events as an alternative to threads to reduce overhead [21]. Interrupt and signal handling mechanisms, core components of any operating system kernel, map into the general event model, where interrupts or signals map into events and their corresponding service routines into event handlers. SPIN, an extensible operating system, relies on events for extensibility [22]. Events in SPIN correspond to procedure calls that can be handled by one or more implementations (event handlers). Like our general model, any number of event handlers can be bound to an event, the order of the handlers can be specified, and handlers can have both bind-time and raise-time parameters. Any number of *guards*, conditions under which the event handler will be executed, can be specified for an event handler. Events and handlers can be synchronous or asynchronous.

Events also form the basis for event notification services ranging from the *selective broadcasting* in Field [25] to standards efforts such as the CORBA Event Notification Service [20]. Such services are typically used in a distributed system for communicating between separate address spaces. Although event and handler profiling makes sense in this context, handler merging may not be feasible since the handlers may need to reside in separate address spaces.

Programming Languages and Toolkits. Programming languages like C#, Java, and Visual Basic support event-based notifications. The use of events in these languages is dual faceted. First, it provides an elegant way of decoupling notification from execution, and second, it adds powerful reactive execution semantics. Events, usually implemented as data structures, represent actions and include a list of handlers implemented as opaque references to function pointers. Handlers are generally *connected* with events and the language runtime is responsible for detecting and notifying programs interested in the occurrence of these events. Although primarily introduced to support GUI-type application semantics, events are now widely used to support extensibility in programs and are increasingly being used in non-GUI contexts.

Qt and GTK, two of the most popular GUI toolkits built on the X-Windows framework, use events to signal the occurrence of actions or user inputs. In the Qt toolkit, events are objects that derive from the QEvent class, while handlers are implemented as virtual functions associated with these event objects. Events signify the occurrence of an action and describe that action; the application responds by calling the Target::event() method, where Target is the target widget/object and event is the corresponding method to be called in response to the event. There are two types of notification, *sendEvent* that provides synchronous activation semantics and *postEvent* that provides asynchronous semantics. Event filters, similar to the *guard* mechanism in SPIN, are used to selectively service events.

In addition to the low-level X-window primitives, GTK provides an elaborate higher-level event mechanism, Signals and Slots, that greatly increases the ease of use and type safety associated with event-based clients. This mechanism is similar to the general event model, where a signal in GTK represents the occurrence of an event and applications use slots to register for notification. Signals are transmitted by the underlying framework and connected to slots, conceptually similar to stubs in application programs. All handlers registered with a slot are invoked on the occurrence of the appropriate signal. 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 to these systems would reduce these costs and make event handling through signals and similar mechanisms less expensive.

6.2 Related Work

Only a small number of papers have addressed compilation oriented optimization of event-based systems. Chambers *et al.* discuss the use of dynamic compilation to optimize event dispatching in SPIN [7]. Unlike our work that uses profile-based static optimizations, this approach relies on optimizations that are carried out during execution. As with other dynamic optimization systems (e.g., Dynamo [2], Tempo [9]), 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. In contrast, the optimizations used here are fairly heavyweight, with a correspondingly higher payoff.

Type feedback [1], which is used to reduce the overheads associated with virtual function calls, is perhaps the closest counterpart in the object-oriented programming language domain. This approach involves monitoring individual call sites at runtime and using this information to predict and in turn optimize likely receiver types.

Also related to our work is research into program specialization of the type done by Synthesis [24] and Synthetix [23] for operating systems. The primary disadvantage of such systems is the resulting loss of portability and maintainability. While the above systems used manual specializations, automated specialization approaches have also been described [17]. Most of these approaches rely on partial evaluation based on knowledge of input values. In contrast, we use program profiling to identify optimizable parts of the program. Our technique can be thought of as "profile directed specialization" and could easily be extended to all classes of programs optimizable through previous specialization efforts.

ILP (Integrated Layer Processing) [8] 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 optimizations do not specifically attempt to reduce memory references related to accessing message data, so ILP could in principle be 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.

Ensemble [12], as mentioned above, uses events to implement communication between layers in a protocol stack. Ensemble protocol stacks are typically deep since they consist of small modules, each of which implements a specific function. Ensemble focuses optimizations on the common sequence of operations that occur in a protocol stack. Each such sequence, called an *event trace*, is triggered by an event such as receipt of a message. The common operations are first annotated by the protocol designer. The code in an event trace is then optimized into one *trace handler* using techniques ranging from eliminating intermediate events to inlining and traditional compiler optimization. Moreover, each event trace has a *trace condition* that must hold for the trace handler to be executed. These conditions are specified using predicates that are again provided by the protocol designer. In comparison, our approach does not require the protocol designer to provide any annotations or predicates. Furthermore, our focus is on providing generally applicable optimization techniques for event-based systems rather than for any one specific system or type of system.

Finally, event-based architectures are gaining popularity in designing complex software systems. Event-based web servers of the SEDA project [27] 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 in some sense the opposite of our approach, which attempts to reduce the number of events in the program. 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, which improves performance by releasing a processor as soon as possible. Our approach gains by using traditional program optimization techniques, such as eliminating procedure calls and other redundant code. Our performance results from figure 10 partially confirm the findings in [27]. When the frame rate is very low, our optimizations actually increase the execution time slightly because locks are held longer by super-handlers, resulting in starvation of other threads even though the CPU utilization is low. As the frame rate and CPU utilization increase, however, the effect of our optimizations becomes more pronounced until a point is reached where system throughput becomes the limiting factor.

7 Conclusion

Event-based programs are used in a variety of domains due to their flexibility, yet have the disadvantage of potentially high performance overhead. Due to the dynamic nature and unpredictability of events—which events occur and when—and event bindings—which handlers are bound to an event when it is raised—compiler optimization techniques have not traditionally been used in this domain. However, in this paper, we have described how such techniques can in fact be applied by exploiting the underlying predictability often exhibited by such programs. Our approach is based on profiling the program to identify commonly occurring event sequences, and then applying a variety of techniques to reduce the overheads associated with such things as indirect function calls and argument marshaling. Experimental results from event-based programs written for Cactus and X Windows suggest that the scope of possible improvement can be significant. Our overall goal in this work is to reduce the overhead of event-based execution to the point where the performance of such programs is competitive with more traditionally structured alternatives.

Acknowledgments

K. Högstedt, W.-K. Chen, and the anonymous referees provided excellent suggestions that improved the paper. P. Bridges provided assistance with the Cactus video program. The work of S. Debray was supported in part by NSF under grants CCR-0073394, EIA-0080123, and CCR-0113633. The work of the other authors was supported in part by DARPA under grant N66001-97-C-8518 and by NSF under grants ANI-9979438 and CCR-9972192.

References

[1] G. Aigner and U. Holzle. Eliminating virtual function calls in C++ programs. In *ECOOP'96 Conference Proceedings, Springer Verlag LNCS 1098*, pages 142–166, 1996.

[2] 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.

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

[4] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain Resort, CO, Dec 1995.

[5] 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.

[6] N. Bhatti and R. Schlichting. A system for constructing configurable high-level protocols. In *Proceedings of SIGCOMM '95*, pages 138–150, Cambridge, MA, Aug 1995.

[7] 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.

[8] 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.

[9] 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.

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

[11] A. Grimshaw, Wm. Wulf, and the Legion team. The Legion vision of the worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, Jan 1997.

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

[13] 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.

[14] 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.

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

[16] 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.

[17] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization tools and techniques for systematic optimization of system software. In *ACM Transactions of Computer Systems, Vol 19, No 2*, pages 217–251, May 2001.

[18] 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.

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

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

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

[22] P. Pardyak and B. Bershad. Dynamic binding for an extensible system. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 201–212, Seattle, WA, Oct 1996.

[23] 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 system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*, pages 314–324, Copper Mountain, CO, Dec 1995.

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

[25] S. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, Jul 1990.

[26] C. Viles, M. Lewis, A. Ferrari, A. Nguyen-Toong, and A. Grimshaw. Enabling flexibility in the Legion run-time library. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 265–274, Las Vegas, NV, Jul 1997.

[27] 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.

[28] 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.