Proceedings of the

ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation ® (PLDI'02)

Berlin, Germany

June 17-19, 2002

Sponsored by the

Association for Computing Machinery
Special Interest Group on Programming Languages

(ACM SIGPLAN)

The Association for Computing Machinery
1515 Broadway
New York, New York 10036

ACM ISBN: 1-58113-463-0

Additional copies may be ordered prepaid from:

ACM Order Department
PO Box 11405
New York, NY 10286-1405

Phone: 1-800-342-6626
(US and Canada)
+1-212-626-0500
(all other countries)
Fax: +1-212-944-1318
E-mail: acmhelp@acm.org

ACM Order Number 548020
Printed in the USA

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.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers; optimization*

General Terms

Profiling, Events, Handlers, 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 [8, 18], form the basis for configurability in systems to build customized distributed services and network protocols [4, 9, 16], are the paradigm used for asynchronous notification in distributed object systems [19], and are advocated as an alternative to threads in web servers and other types of system code [20, 23]. 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 [5, 14]. 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 a path profile through the call graph of the program. These profiles are then used to identify commonly encountered events, 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. 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 [2] 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 [10, 12]. The third is a client side tool that uses X Windows, a popular system for building GUIs [18]. This is followed by discussions of possible extensions in section 5 and related 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 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 program.

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 runtime *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 to 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.

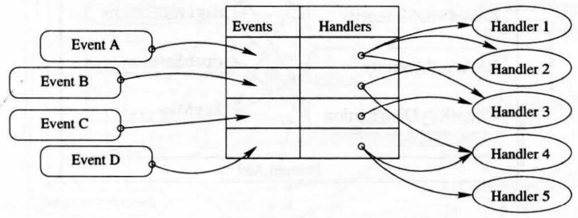

Figure 1: Event bindings

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 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.

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 a 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

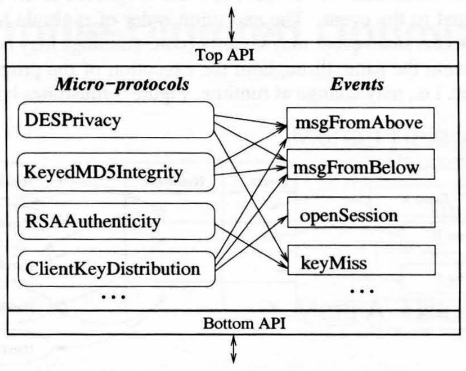

Figure 2: Cactus composite protocol

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 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 [10]. 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 4 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.

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" [18]. 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. OPTIMIZATION APPROACH

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 profiles). Event-based systems, in contrast, are largely unpredictable in their runtime behavior due to the uncertainties associated with the be-

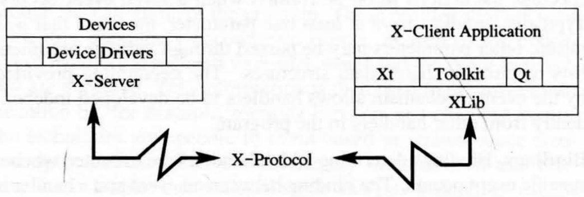

Figure 3: Architecture of X Window systems

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.