Causeway: Support for Controlling and Analyzing the Execution of Multi-Tier Applications

Anupam Chanda[1], Khaled Elmeleegy[1],
Alan L. Cox[1], and Willy Zwaenepoel[2]

[1] Rice University, 6100 Main Street, Houston, Texas 77005, USA
{anupamc,kdiaa,alc}@cs.rice.edu
[2] Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland
willy.zwaenepoel@epfl.ch

Abstract. Causeway provides runtime support for the development of distributed *meta-applications*. These meta-applications control or analyze the behavior of multi-tier distributed applications such as multi-tier web sites or web services. Examples of meta-applications include multi-tier debugging, fault diagnosis, resource tracking, prioritization, and security enforcement.

Efficient online implementation of these meta-applications requires metadata to be passed between the different program components. Examples of metadata corresponding to the above meta-applications are request identifiers, priorities or security principal identifiers. Causeway provides the infrastructure for injecting, destroying, reading, and writing such metadata.

The key functionality in Causeway is forwarding the metadata associated with a request at so-called transfer points, where the execution of that request gets passed from one component to another. This is done automatically for system-visible channels, such as pipes or sockets. An API is provided to implement the forwarding of metadata at system-opaque channels such as shared memory.

We describe the design and implementation of Causeway, and we evaluate its usability and performance. Causeway's low overhead allows it to be present permanently in production systems. We demonstrate its usability by showing how to implement, in 150 lines of code and without modification to the application, global priority enforcement in a multi-tier dynamic web server.

1 Introduction

Many applications, e.g., web sites generating dynamic content and web service applications, have multi-tiered implementations. A multi-tier application is composed of multiple program components communicating among themselves to execute incoming requests. In such applications, a request is executed by multiple threads of control on different application components, the threads of control exchanging data among themselves along communication channels. For example, an application may be composed of a web server, an application server, and a database server: requests are executed by all three programs communicating with each other to exchange request data.

Often, systems to control or analyze the execution of multi-tier applications are written to perform tasks like multi-tier debugging, fault diagnosis, resource tracking, prioritization, and security enforcement. Examples include Pinpoint [5], Magpie [4, 9], and Domain and Type Enforcement (DTE) [3] for Unix systems. We term these and similar systems that control or analyze the execution of multi-tier applications as *meta-applications*.

Traditionally, there have been two approaches to writing such meta-applications: a log-based approach, and a metadata-passing approach. The log-based approach operates in two phases — first, execution events of the application are recorded in logs, and next, the log records are analyzed. Magpie [4, 9] and TraceBack [2] are examples of systems employing this approach. The log-based approach cannot affect the execution of requests in an online manner because processing of a log record lags the corresponding execution event by a positive time delay. Additionally, the execution events on the different tiers belonging to the same request need to be identified and connected while processing the log records.

The metadata-passing approach propagates *metadata* — arbitrary, out-of-band data — in addition to request data along execution paths. The meta-application accesses and utilizes this metadata to achieve its goal. Often, the metadata also serves in connecting a request's execution events spread across the tiers of the system, e.g., if it contains a request identifier. Several examples of meta-applications using this approach exist in the literature, e.g., Pinpoint [5] and DTE [3]. Pinpoint and DTE use request identifiers and security principal identifiers as metadata respectively. These meta-applications use hand-crafted code to handle and propagate metadata.

Unlike the log-based approach, the metadata-passing approach can affect the execution of requests in an online manner, e.g., Real-Time CORBA [10] which propagates priorities among application components to affect scheduling. Hence, we adopt the metadata-passing approach to building meta-applications. Our objective is to provide a framework that makes development of meta-applications using this approach easier.

In this paper we introduce Causeway, a framework to facilitate the association and propagation of metadata along request execution paths in a multi-tier application. Causeway provides an interface to associate metadata with threads of control and facilitates the propagation of metadata across communication channels. Causeway aids the development of meta-applications by performing all necessary management to handle and propagate metadata. This obviates the need for hand-crafted code for the common requirements of different meta-applications employing the metadata-passing approach.

The alternative to Causeway, propagating metadata at application level, involves augmenting all application-level inter-process communication protocols — a tedious solution. By making propagation of metadata a system-level function, it becomes independent of the application-level communication protocol used. Further, in a multi-tier application, it is possible that some individual components are unaware of the presence of metadata or choose to ignore it. Consider a three-tier system, where the middle tier component is unaware of metadata. The front and the back-end tiers may still, however, need to access metadata. In this scenario, system support for metadata propagation is required in the middle tier.

Causeway performs automatic propagation of metadata across *system-visible* communication channels. Such channels are those implemented in the operating system kernel and system libraries, e.g., pipes and sockets. Augmented kernel and system libraries provide Causeway's support for system-visible channels. Causeway provides an API to be called from application code to perform metadata propagation across *system-opaque* channels, e.g., shared memory. Support for system-opaque channels is the essential difference between Causeway and Stateful Distributed Interposition (SDI) [11].

We have implemented a prototype of Causeway, measured its overhead, and built a useful meta-application using Causeway. We summarize our experience with Causeway as follows:

- Adding support to propagate metadata across system-visible channels required modest effort.
- The measured overhead of Causeway to propagate metadata was small in absolute cost (order of microseconds) and it scaled well with increasing metadata size. The overhead of Causeway, while not propagating any metadata, was insignificant — less than 3% for a microbenchmark involving the pipe channel. Thus Causeway may reasonably remain a part of a production environment whether implementing a meta-application or not.
- Using Causeway we could rapidly implement a distributed priority enforcement system where the priority of a request is injected and propagated as metadata, and accessed to implement global priority scheduling. This required writing only about 150 lines of code on top of Causeway to change the priority of threads executing requests. We evaluated this system on an implementation of the TPC-W [12] benchmark.

The rest of the paper is organized as follows. We describe the design of Causeway in Section 2. In Section 3 we measure Causeway's overhead with two microbenchmarks. In Section 4 we evaluate Causeway's complexity to support system-visible channels, and measure the overhead of Causeway on an implementation of the TPC-W benchmark. We describe the distributed priority enforcement system using Causeway in Section 5. Related work is covered in Section 6. We conclude in Section 7.

2 Causeway Design

At an abstract level, Causeway works as follows. A request to an application is executed by one or more threads of control, possibly in one or more tiers. Threads exchange request data along communication channels, e.g., sockets, pipes and shared memory. Causeway's interface supports injection, inspection, modification and removal of metadata. Metadata is assigned to a thread when it performs injection. When a thread sends request data to another thread along a channel, Causeway transfers metadata from the former thread to the latter. Support for metadata propagation is required at *transfer points* where an application thread sends to or receives data from a channel. In this way, metadata, once injected, is propagated along the request execution paths.

Causeway has two parts: (1) a set of interfaces that are used by applications to manage and utilize metadata, and (2) mechanisms that implement propagation of metadata. First, we describe the structure and composition of metadata.

2.1 Metadata

Metadata in Causeway consists of a two-tuple containing the metadata type and the metadata value. Examples of metadata types include request priority, request identifier, and security principal identifier. Meta-applications can define new metadata types, if required.

2.2 Interfaces

Meta-applications can interact with Causeway in two ways — through an interface to inject and access metadata and through a callback interface in which Causeway calls handlers registered by the meta-application.

Metadata Interface Causeway provides interfaces for injection, inspection, modification, and removal of metadata. These interfaces may be called from user-level or kernel-level.

`int cw_type_query(void *addr, int types[], int ntypes)`
`int cw_data_lookup(void *addr, int mtype, struct cw_metadata *md_p)`
`int cw_data_insert(void *addr, int mtype, struct cw_metadata md)`
`int cw_data_remove(void *addr, int mtype)`

Table 1. The Causeway API

Causeway manages metadata in a dictionary keyed by the address of the associated *entity*. An entity is either a thread of control or data that is read from or written to a channel. A thread's metadata is propagated to the data written on a write operation, subsequently this metadata is propagated from the data to a thread performing a read operation. Further, a thread can remove metadata associated with itself or a data entity. Table 1 shows the function signatures of the Causeway API. The Causeway API performs metadata operations in the following manner:

- cw_type_query retrieves the collection of *all* metadata types associated with addr in the types array of size ntypes. On successful completion, cw_type_query returns the number of metadata types retrieved and -1 on error. The types array must be large enough to hold all the metadata types associated with addr otherwise an error is flagged.
- cw_data_lookup retrieves the metadata of type mtype associated with addr. It returns 0 on successful completion and -1 on error.
- cw_data_insert inserts the given metadata md of type mtype and associates it with addr, overwriting any prior metadata of that type. It returns 0 on successful completion and -1 on error.
- cw_data_remove removes any existing metadata of type mtype associated with addr. It returns 0 on successful completion and -1 on error.

Callback Interface Using Causeway's callback interface the meta-application can register a *transfer-point* callback method. A transfer point is a point where data is read from or written to a channel by a thread. At a transfer point Causeway determines if the type of the metadata being passed has a callback method registered. If a callback method exists, it is invoked with the metadata as an argument. The callback method reads and possibly modifies the metadata. The callback method can call arbitrary operating system code, e.g., to change the priorities of threads.

The signatures of a callback method and the callback interface are shown in Table 2. A callback method is of type callback_t. The callback interface, reg_callback_method, registers a given callback method for a given metadata type at a transfer point.

typedef void (*callback_t)(struct cw_metadata **md, int mtype);	
callback_t	callback_method;
void	reg_callback_method(int mtype, callback_t callback_method);

Table 2. The Callback Interface

2.3 Support for Propagation of Metadata

When a thread performs a write on a channel, the thread's metadata is associated with the data written into the channel. On a subsequent read on the channel by a thread, metadata is propagated from the data and assigned to the thread.

There are two ways metadata can be assigned to a thread — injection and propagation across a channel. Newly assigned metadata replaces the thread's existing metadata of the same type.

Transfer Points Places where a thread writes to or reads from a channel are transfer points. Channels are of two types: system-visible channels that occur in the operating system kernel and system libraries, e.g., sockets and pipes, and system-opaque channels that occur in the application, e.g., shared memory. Causeway exports a Systems Programming Interface (SPI) consisting of a single function `cw_metadata_xfer` for the purpose of implementing transfer points. `cw_metadata_xfer` takes a source entity and a destination entity as arguments. It obtains the source entity's metadata and assigns the obtained metadata to the destination entity. At a transfer point for either a system-visible or system-opaque channel, a single call to `cw_metadata_xfer` is performed.

2.4 System-visible Channels

For system-visible channels, the metadata transfer SPI is automatically called from an augmented kernel and system libraries to implement Causeway's support for metadata propagation. Sockets and pipes are system-visible channels supported by Causeway. Further, for a multi-threaded program, metadata needs to be propagated between the user-level thread and the kernel-level thread on entry to and exit from the kernel because multiple user-level threads may be multiplexed on top of a kernel-level thread. Metadata propagation between a user-level thread and a kernel-level thread constitutes additional system-visible channels in Causeway. We enumerate below the transfer points for system-visible channels:

1. *User-level thread to kernel-level thread:* On entry to the kernel, Causeway transfers metadata from the user-level thread to the kernel-level thread running it.
2. *Kernel-level thread to user-level thread:* On exit from the kernel, Causeway transfers the kernel-level thread's metadata to the user-level thread.
3. *Kernel-level thread to message:* When a kernel-level thread writes a message on a socket or a pipe, its metadata is transferred to the message.
4. *Message to kernel-level thread:* When a kernel-level thread receives a message from a socket or a pipe, metadata is transferred from the received message to the kernel-level thread.

These transfer points occur in the operating system kernel and the threading library.

Causeway handles sockets and pipes similarly. When a thread writes to a socket (or a pipe), Causeway associates metadata from the thread to the data written via the metadata transfer SPI described above. Similarly, on a subsequent read from the socket by another (or the same) thread, metadata is propagated from the data to the thread.

The above applies for `LOCAL` sockets only. For `INTERNET` sockets, data is encapsulated in IP packets for send and receive across sockets. Causeway encapsulates metadata, in addition to data, in the IP packets. For IPv4, Causeway encapsulates metadata in the IP header as IP options. In particular, Causeway defines a new IP option type, populates the IP header with the option type, length, and payload. At the receiver side, metadata, if any, is extracted from the received IP options. Since IP options can be a maximum of 40 bytes only, with 1 byte each for the type and length fields, via this mechanism Causeway can transfer at most 38 bytes of metadata in IP packets. This limit on metadata size is deemed enough for most practical purposes. This limitation is an artifact of Causeway's implementation and not its design. A general purpose tunneling protocol could be used to overcome this limitation, if required. For IPv6, Causeway uses the destination options in the IP header which does not have any size limitation. Further details about that are outside the scope of this paper.

2.5 Shared Memory — System-opaque Channel

For system-opaque channels, the application must be modified to call the metadata transfer SPI to perform propagation of metadata. Causeway supports metadata propagation across shared memory — a system-opaque channel implemented in user-space. A transfer point needs to be inserted in the application where a user-level thread reads from or writes to shared memory. Producer-consumer is a popular model of shared memory usage. At an abstract level, the model works as follows. Producers and consumers share a buffer or queue of objects. A producer creates an object, acquires a lock to enter the critical section, adds the object to the shared buffer or queue, and releases the lock. A consumer acquires a lock to enter the critical section, retrieves and removes an object from the shared buffer or queue, releases the lock, and then accesses the retrieved object. The use of system-supported synchronization primitives, like `pthread_mutex` or `pthread_rwlock`, simplifies the task of identifying the producer-consumer communication channels through shared memory.

Two transfer points, one in the producer code and the other in the consumer code are inserted. Both transfer points use the metadata transfer SPI. The producer transfer point associates the producer thread's metadata with the produced object. The consumer transfer point retrieves the metadata associated with the consumed object and propagates it to the consumer thread. Causeway provides a user-level library that exports the metadata transfer SPI and manages the metadata associated with shared memory objects.

2.6 Heterogeneity of Operating System Kernel and Hardware

It is quite common for a multi-tier application to be spread across machines running heterogeneous operating system kernels on diverse hardware platforms. The design of Causeway mandates that all inter-machine metadata propagation be typed and be transmitted in network byte order. This ensures correct interpretation of metadata at the receiver. Further, our implementation of Causeway in FreeBSD lays out a blueprint for its implementation in other operating system kernels. In Section 4.1 we list the transfer points in the FreeBSD kernel required for the system-visible channels. An equivalent set of transfer points is required in another operating system kernel, such as Linux.

2.7 Operating System Specific Meta-applications

Sometimes, parts of a meta-application may require modifications to the operating system kernel. Under such circumstances, the meta-application becomes operating system specific. For example, we implemented a distributed priority enforcement system on top of Causeway which may alter priorities of threads and processes in a system — an operating system specific task. Thus, this meta-application is operating system specific. On the other hand, if all we wanted in a meta-application is to tag identifiers with requests, it would require no operating system modification other than Causeway itself.

3 Microbenchmarks

In this section we quantify the overhead imposed by our implementation of Causeway at the transfer points for two system-visible channels. We chose light-weight applications to provide maximum exposure to Causeway's overhead. We wrote two microbenchmarks: the first measuring the overhead associated with the transfer points for metadata propagation between a user-thread and a kernel thread, and the second measuring the overhead for the transfer points for the pipe channel.

In the first microbenchmark, a process creates a `pthread` which invokes a `getpid` call. This test brings out the cost of metadata propagation across the user-kernel boundary, because on each entry to and exit from the kernel, metadata is transferred from user space to kernel and vice versa. We repeat the `getpid` call multiple times and measure its average cost. We perform this experiment under the following scenarios: (1) without inserting the transfer point, which is the base case, (2) inserting the transfer point but transferring 0 bytes of metadata, (3) transferring 1 byte of metadata, and (4) transferring 32 bytes of metadata.

Description	Cost (machine cycles)	Cost (microseconds)	Overhead (%)
Base case	7001	2.92	-
0 byte metadata	7841	3.27	12.0
1 byte metadata	9369	3.90	33.8
32 bytes metadata	9409	3.92	34.4

Table 3. Causeway Overhead (`getpid` test)

Table 3 shows the results of the above experiment. The cost of `getpid` increased by about 840 machine cycles when a transfer point was introduced. We used a 2.4 GHz Pentium 4 Xeon, so this overhead translates to about 0.35 microseconds. This result shows the cost of having the Causeway framework but not using it to propagate any metadata. The overhead increased by about 1500 machine cycles or about 0.6 microseconds when transferring 1 byte of metadata. To transfer 32 bytes of metadata, the further increase in overhead was small: about 40 machine cycles or 0.02 microseconds. In relative terms, the overhead with respect to the base case ranged from about 12% to less than 35% to transfer metadata in the above test.

Description	Cost (machine cycles)	Cost (microseconds)	Overhead (%)
Base case	35782	14.9	-
0 byte metadata	36807	15.3	2.9
1 byte metadata	49858	20.8	39.3
32 bytes metadata	54383	22.66	52.0

Table 4. Causeway Overhead (pipe test)

The results of the above experiment show that the overhead of using Causeway is small. The overhead of inserting a transfer point is less than half of a microsecond. The overhead of transferring 32 bytes of metadata is about 1 microsecond, and the overhead scales well with increasing metadata size.

The second microbenchmark measures the cost of transferring 1 byte of data between two processes across a pipe. As before, we perform this experiment under the four scenarios used in the previous microbenchmark. Table 4 shows the result for the pipe test. The overhead of inserting a transfer point but passing no metadata is similar to that of the `getpid` test. The overhead of passing metadata is higher because the metadata is propagated across address spaces. Nevertheless, the overhead of propagating up to 32 bytes of metadata is less than 8 microseconds, a small amount. Finally, the overhead scales well with increasing metadata size. In this test Causeway's overhead ranged from less than 3% to about 52% over the base case.

Note that for the above measurements we could not use a microbenchmark consisting of a network server and client as the cost of sending messages over the network is several orders of magnitude higher than the overhead of Causeway in terms of absolute cost and we would not have been able to detect the overhead of Causeway with such a microbenchmark.

4 Evaluating Causeway

In this section we quantify the complexity involved in Causeway to insert transfer points for system-visible channels, and transfer points in an implementation of the TPC-W [12] benchmark. We also measure Causeway's overhead on TPC-W.

4.1 Transfer Points for System-visible Channels

Sockets, pipes, and user-level thread/kernel-level thread boundary are the system-visible channels supported by Causeway. Six transfer points in the FreeBSD 5.2

kernel support metadata propagation across these channels as shown in Table 5. The user thread to kernel thread and kernel thread to user thread transfer points are required if the application is multithreaded. The socket and pipe transfer points are required if the application performs interprocess communication. Transfer points within system-visible channels do not require reimplementation for each new application.

Location	Description	File name	Function name
Kernel	User thread to kernel thread	`kern/kern_kse.c`	`thread_user_enter`
Kernel	Kernel thread to user thread	`kern/kern_kse.c`	`thread_userret`
Kernel	Kernel thread to socket message	`kern/uipc_socket.c`	`sosend`
Kernel	Socket message to kernel thread	`kern/uipc_socket.c`	`soreceive`
Kernel	Kernel thread to pipe message	`kern/sys_pipe.c`	`pipe_write`
Kernel	Pipe message to kernel thread	`kern/sys_pipe.c`	`pipe_read`

Table 5. Transfer Points for System-visible Channels in the FreeBSD Kernel

4.2 Transfer Points for Apache and MySQL

We used Causeway to propagate metadata in an implementation of the TPC-W [12] benchmark. Our implementation of the TPC-W benchmark used the Apache web server (version 1.3.31) built with the PHP module (version 4.3.6) and the MySQL database server (version 4.0.16). The TPC-W interactions are implemented as PHP scripts.

Apache is a multi-process web server and does not use shared memory communication among the different processes. Thus, no transfer points are required in Apache.

MySQL is a multi-threaded program and it uses the `libpthread` library on FreeBSD. Inspection of the MySQL source code revealed that though individual MySQL `pthread`s access some shared data structure in a synchronized manner, there is no communication between threads to exchange data corresponding to a single request. In other words, a request in MySQL is executed in its entirety by a single `pthread`. An incoming database connection is accepted by a listener thread and handed over to a worker thread. The worker thread reads the request, executes it and sends back the response. Hence, no transfer points are required in MySQL as well.

In TPC-W, Apache and MySQL exchange messages across sockets. MySQL uses user-level thread on top of kernel-level threads. Thus Causeway's support for metadata propagation across system-visible channels, viz., sockets, and user-level thread and kernel-level thread boundary, suffices for our implementation of TPC-W using Apache and MySQL. This support is provided in an augmented FreeBSD kernel.

4.3 Overhead of Causeway on TPC-W

We conducted an experiment to evaluate the overhead imposed by Causeway on our implementation of TPC-W under a realistic workload. We subjected TPC-W to a workload consisting of emulated clients exercising the *shopping mix* [12] workload. Apache, MySQL and the load generator ran on separate machines. All the machines were 2.4 GHz Pentium Xeon with 2 Gigabytes of memory, and were connected by switched Gigabit ethernet. We varied the number of concurrent emulated clients and measured the throughput (interactions per minute) obtained from TPC-W. We compare the throughput obtained with the Causeway framework with that obtained without the Causeway framework (base case). Under Causeway we transferred 4 bytes of metadata across each transfer point for TPC-W. Table 6 shows the results of this experiment; Causeway's overhead on TPC-W's throughput remains less than 5%, further it does not increase with increasing load on the system and remains fairly constant. This result shows that Causeway may be used in a production environment without any substantial performance degradation.

No. of concurrent emulated clients	Throughput (base case)	Throughput using Causeway	Causeway Overhead(%)
10	89.4	89	4.91
50	424.8	411	3.25
100	844.2	826.4	2.11

Table 6. TPC-W Throughput (interactions/minute) for Shopping Mix

5 Example Use of Causeway: Multi-tier Priority Propagation

Meta-applications to control and analyze the execution of applications can be built easily using Causeway. We illustrate one such meta-application here.

Using Causeway we could rapidly implement a priority propagation system, enabling a multi-tier application to prioritize the execution of requests. Under this system, upon receiving a request the application injects a priority as metadata, Causeway propagates this priority metadata with the execution of the request to each of the tiers, and the meta-application uses the priority metadata to enforce priority scheduling on each tier. The meta-application is automatically invoked on each tier by Causeway's transfer point callback mechanism.

The implementation of the multi-tier priority propagation system on top of Causeway required writing about 150 lines of code. We tested the multi-tier priority propagation system with an implementation of the TPC-W benchmark [12]. No modifications were required in the TPC-W application code, other than the injection of priority metadata.

5.1 Metadata Access

The priorities are injected into the system when a request arrives, using the metadata access API of Causeway. We register transfer point callback methods at the transfer points from a kernel thread to a user thread, and from a socket to a kernel thread. These callback methods change the priorities of the user thread and the kernel thread respectively. The first callback method affects the scheduling of MySQL `pthread`s while the second one achieves the same for Apache processes.

5.2 Application

We use the TPC-W [12] benchmark as our application. TPC-W simulates an online bookstore. Its implementation consists of a front-end web server, providing an HTTP front-end and serving static content, a middle-tier application that implements the business logic, and a back-end database server that stores the dynamic content of the site. The benchmark defines 14 interactions with the web site, 13 of which access the database. 6 interactions write to the database, while the others are read-only. Our hardware and software platforms are the same as described earlier in Section 4.

5.3 Experiment

The goal of the experiment is to demonstrate that multi-tier priority propagation using Causeway, without application modification, has considerable benefits. Our performance metric is the response time of the high-priority requests. We show that the response time of high priority requests is relatively independent of the load imposed on the system. We also demonstrate that enforcing priority at both tiers (web server and database server) is superior to only enforcing it at the first tier.

We define a foreground load as a sequence of 100 instances of each TPC-W interaction, spaced out in time by one second. We define a background load that directs a steady stream of read-only requests at the site. The background load simulates visitors browsing the web site, while the foreground load simulates customers performing the actions that may lead to purchases at the site, thereby deserving higher priority. We use two different levels of background load: one which overloads the system and one which imposes a moderate load without, however, saturating the system.

We have two levels of priority in the system: a default priority and a high priority. Requests originating from the background load are always tagged with metadata indicating the default priority. To demonstrate the effect of priorities, we perform two experiments, with requests from the foreground load tagged with metadata either indicating the high priority or the default priority. In addition, to demonstrate the difference between single-tier and multi-tier priority enforcement, we run an experiment in which on the web server the priorities are enforced by the transfer point callback methods as described above, but on the database server they are ignored.

5.4 Results

Table 7 shows the average response times (along with the 95% confidence intervals) in milliseconds for each of the interactions under the following conditions:

1. No background load: This case shows the baseline response time for each interaction.
2. No priority: The background load is present, but neither of the tiers enforce priority scheduling based on the metadata.
3. Priority in first tier: The background load is present, and the first tier (the web server) enforces priority scheduling based on the metadata.
4. Priority in both tiers: The background load is present, and both tiers enforce priority scheduling based on the metadata.

As further illustration of the results, we show in Figure 1 the response times, sorted in descending order, for the execution of the 100 requests of the search-request interaction under the four cases as described above.

Inter-action	No back-ground load	No priority	Priority in 1st. tier	Priority in all tiers
admin-confirm	60 (±0.2)	1936 (±3.8)	1993 (±38)	342 (±71)
admin-request	59 (±0.01)	1617 (±120)	868 (±85)	68 (±13)
best-sellers	918 (±49)	3173 (±986)	3016 (±234)	940 (±33)
buy-confirm	85 (±1.3)	1951 (±36)	1992 (±67)	1457 (±131)
buy-request	60 (±1)	1930 (±4.5)	1915 (±59)	81 (±36)
customer-reg	55 (±1.2)	931 (±88)	61 (±1.5)	60 (±1.6)
home	61 (±1.7)	1737 (±93)	1095 (±102)	63 (±2.2)
new-product	81 (±1.7)	1933 (±3)	1969 (±28)	85 (±4)
order-display	60 (±0.8)	1930 (±3)	1970 (±4)	64 (±4)
order-inquiry	40 (±0.01)	42 (±2.2)	40 (±1)	40 (±0.3)
product-detail	60 (±0.6)	1516 (±127)	966 (±100)	68 (±14)
search-request	60 (±0.03)	1533 (±127)	987 (±102)	61 (±0.7)
search-result	670 (±0.6)	2628 (±314)	2528 (±5.3)	671 (±1.5)
shopping-cart	70 (±0.9)	1931 (±4)	1984 (±6)	217 (±40.5)

Table 7. Average Response Time and 95% Confidence Interval (in milliseconds) for the TPC-W Interactions under High Background Load

Table 7 and Figure 1 reflect the behavior under a background load that pushes the system into overload. The same results for a moderate background load are shown in Table 8 and Figure 2.

5.5 Discussion

The results overall confirm the benefits of multi-tier priority enforcement. With priorities enforced at both tiers the response times approximate those under no

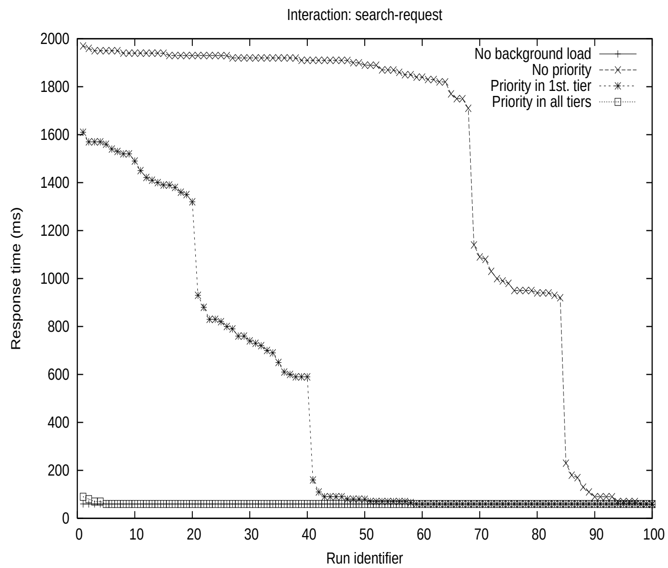

Fig. 1. Response Time Distribution (Sorted in Descending Order) for Search-Request Interaction (High Background Load)

Inter-action	No back-ground load	No priority	Priority in 1st. tier	Priority in all tiers
admin-confirm	60 (±0.2)	95 (±6)	90 (±6)	65 (±1.3)
admin-request	60 (±0.2)	92 (±6)	65 (±2.7)	60 (±0.15)
best-sellers	918 (±49)	1092 (±165)	1137 (±158)	912 (±0.9)
buy-confirm	85 (±1.3)	136 (±6)	123 (±6)	94 (±1.8)
buy-request	60 (±1)	103 (±7)	99 (±6)	63 (±1.7)
customer-reg	55 (±1.3)	78 (±4.4)	62 (±2.6)	59 (±1.1)
home	61 (±1.9)	98 (±6.2)	82 (±5.5)	62 (±2)
new-product	81 (±1.7)	125 (±9.6)	101 (±7)	84 (±3.4)
order-display	60 (±0.8)	102 (±6.9)	101 (±6.5)	62 (±1.5)
order-inquiry	40 (±0.01)	40 (±0.15)	40 (±0.01)	40 (±0.01)
product-detail	60 (±0.6)	94 (±6)	64 (±2.4)	60 (±0.2)
search-request	60 (±0.04)	97 (±6.3)	65 (±2.8)	60 (±0.14)
search-result	670 (±0.62)	715 (±19.7)	728 (±11.8)	667 (±3.2)
shopping-cart	70 (±0.86)	110 (±6.2)	83 (±4.1)	73 (±1.1)

Table 8. Average Response Time and 95% Confidence Interval (in milliseconds) for the TPC-W Interactions under Moderate Background Load

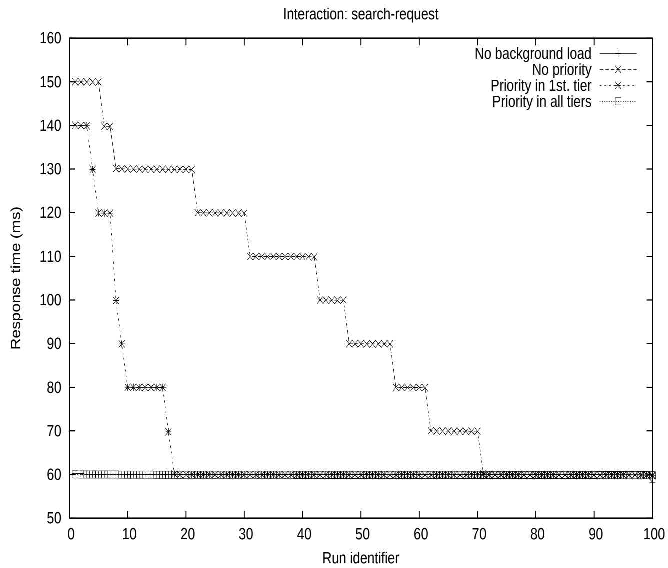

Fig. 2. Response Time Distribution (Sorted in Descending Order) for Search Request (Moderate Background Load)

load, and they are substantially better than those in the absence of priorities or in the presence of priorities only at the first tier. The results for single-tier priority enforcement are better than with no priorities, but inferior to using priorities at both tiers. The differences are more outspoken in the case of overload, but remain present even under more moderate loads. Given that Causeway allows multi-tier priority propagation without modification of the application and without noticeable overhead, we argue that this serves as a convincing demonstration of its merits.

More detailed inspection of the results on a per-interaction basis leads to some additional observations. First, in looking at Table 7 we see that for a large number of the interactions the response time under load with multi-tier priorities is almost identical to the response time under no load. For a few interactions, however, the response under load is higher, even with the priorities. This observation is explained by the fact that the background load acquires read locks on a certain table in the database, and the fact that the interactions that show a slowdown under load acquire an exclusive lock on that table. As a result, independent of priorities, the foreground interactions need to wait for all current readers to finish before they can proceed at the database. Under overload, there can be a large number of such reads in progress, explaining the marked increases in response time for the admin-confirm, buy-confirm, buy-request and shopping-cart interactions. For the moderate load where only a very few such readers are present, the differences almost vanish (see Table 8). For foreground interactions that have no conflicts with the background load, there is almost no difference between the the no-load case and the case of load and with multi-tier priorities.

Second, in a few cases, namely the customer-registration and the order-inquire interactions, there is no difference between single-tier and multi-tier priorities. This is the result of the fact that for these interactions there is no access to the database or the cost is mainly governed by application execution and not by database access. Conversely, for the interactions whose cost is primarily governed by database access or for the interactions that acquire exclusive locks on the database, there is a more pronounced difference between single-tier and multi-tier priorities. In these cases, the benefit of enforcing priority at the first tier is also limited relative to the case of not having priorities at all.

6 Related Work

Several meta-applications to control or analyze multi-tier applications exist in the literature. The use of request tagging has been utilized to determine faults in Internet services [5]. The resulting Pinpoint system uses instrumentation of the J2EE platform to pass on request identifiers among the different components of the system. Each component registers information in a log about the request identifier, the component identifier and whether a particular operation results in success or failure. Failure is defined as throwing a Java exception, a runtime exception, an infinite loop, etc. The log is statistically analyzed using data clustering techniques to find faulty components. Pinpoint does not support applications spanning multiple machines, but the authors state that the Java RMI libraries can be extended to pass request identifiers across machines. Unlike Causeway, Pinpoint does not track execution events in the kernel as its instrumentation does not extend beyond the J2EE platform.

Aguilera et al. [1] infer causal paths from message traces to locate nodes causing performance bottlenecks; their implementation is based on the Pinpoint system [5]. They collect traces of messages between nodes, process them offline to find causal relationships among them, and study the delay patterns of the messages to infer which node is causing the bottleneck. Their system is intended to operate in a "black-box" environment, and therefore tries to be minimally invasive. Causeway is more invasive, requiring kernel and library changes, but in turn provides more functionality. In particular, it's deterministic rather than being heuristic, and much more fine-grained.

Magpie [4, 9] logs events, and extracts events belonging to a particular request execution by performing temporal joins over the log of events. These joins are based on application-specific schemas, which may require considerable expertise and knowledge about the application. Magpie and request identification using Causeway present an interesting set of tradeoffs. Magpie does not require kernel or library modifications, and leverages event logging facilities already present in Windows. In contrast, Causeway accepts the premise of such modifications, and as a result avoids the need for detailed knowledge about the application.

TraceBack [2] provides a debugging facility in production systems. It can identify what first went wrong in the event of a program crash, hang or exception. It instruments the program to record control flow information at runtime, which is later analyzed to locate the occurrence of the first fault.

DTE [3] propagates domain and type information among communicating processes providing security and access control for interprocess communication. While DTE provides security mechanisms, Causeway may be used to implement arbitrary meta-applications.

Perhaps the work closest to Causeway is Stateful Distributed Interposition (SDI) [11] which propagates contextual information along request execution paths in a multi-tier application. Resource constraints and security classification are examples of contextual information. Contextual information in SDI and metadata in Causeway are analogous. SDI assumes all communication channels in a multi-tier program to be system-visible, and thus it does not propagate contextual information across system-opaque channels. Causeway supports metadata propagation across shared memory, a system-opaque channel.

7 Conclusions

We have designed Causeway, operating system support for facilitating development of meta-applications to control and analyze multi-tier applications. Causeway provides interfaces for metadata injection and access which can be used for propagation of metadata in multi-tier applications. Propagated metadata can be accessed and used to implement the desired meta-application. We have implemented Causeway in the FreeBSD operating system kernel. The complexity of adding transfer points in the FreeBSD kernel for system-visible channels was modest. Causeway's support for system-visible channels suffices for metadata propagation in an implementation of the TPC-W [12] benchmark using Apache and MySQL — no modification to Apache or MySQL was required. We measured the overhead of Causeway and found it small enough so that it can be used in a production environment. Further, the overhead scales well with increasing metadata size and load on the application. We have demonstrated the use of Causeway by implementing a multi-tier priority enforcing system and using it to achieve global priority enforcement on our implementation of the TPC-W benchmark. This required adding only about 150 lines of code on top of Causeway.

As ongoing and future work we are implementing call path profiling of distributed programs on top of Causeway. Call path profiling [7, 8] associates resource consumption of program execution with call paths. At any point in the program execution, a call path is defined as the sequence of call sites used to activate each of the procedure frames on the call stack when the given point of execution is reached. Call path profilers are superior to call-graph profilers like `gprof` [6] because they can distinguish resource consumption of a procedure based on the call paths leading to it.

In a distributed program whose components perform Remote Procedure Calls (RPCs) among themselves, we can use Causeway to propagate the context information (call path) from the caller to the callee, and use this propagated context information to annotate the callee's profiles. Profiles of the caller and the callee may then be stitched together in a single call path tree using these annotations. The end result is an end-to-end call path profile of a distributed program — such a profiler does not exist in the literature. This profiling system illustrates another useful meta-application on top of Causeway.

References

1. M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 74–89, Oct. 2003.
2. A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel. TraceBack: First Fault Diagnosis by Reconstruction of Distributed Control Flow. In *Conference on Programming Language Design and Implementation (PLDI) 2005*, pages 201–212, June 2005.
3. L. Badger, D. F. Sterne, D. L. Sherman, K. M. Walker, and S. A. Haghighat. A Domain and Type Enforcement UNIX Prototype. In *Fifth USENIX UNIX Security Symposium*, June 1995.
4. P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, pages 259–272, Dec. 2004.
5. M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the International Conference on Dependable Systems and Networks (IPDS Track)*, pages 595–604, June 2002.
6. S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.
7. R. J. Hall. Call path profiling. In *Proceedings of the 14th International Conference on Software Engineering*, pages 296–306, 1992.
8. R. J. Hall and A. J. Goldberg. Call path profiling of monotonic program resources in UNIX. In *Proceedings of the USENIX Summer Technical Conference*, 1993.
9. R. Isaacs, P. Barham, J. Bulpin, R. Mortier, and D. Narayanan. Request extraction in Magpie: events, schemas and temporal joins. In *SIGOPS EW'04: ACM SIGOPS European Workshop*, Sept. 2004.
10. Jon Currey. Real-Time CORBA Theory and Practice : A Standards-based Approach to the Development of Distributed Real-Time Systems. At http://www.uninova.pt/~jmf/aptr/Documentos/CorbaRT.pdf.
11. J. Reumann and K. G. Shin. Stateful Distributed Interposition. *ACM Transactions on Computer Systems*, 22(1):1–48, Feb. 2004.
12. T. P. P. C. (TPC). TPC BENCHMARK W (web commerce). At http://www.tpc.org/tpcw/, Feb. 2002.