Memory-manager/Scheduler Co-design: Optimizing Event-driven Servers to Improve Cache Behavior

Sapan Bhatia
INRIA/LaBRI
sapan.bhatia@inria.fr

Charles Consel
INRIA/LaBRI
charles.consel@inria.fr

Julia Lawall
DIKU, University of Copenhagen
julia@diku.dk

Abstract

Event-driven programming has emerged as a standard to implement high-performance servers due to its flexibility and low OS overhead. Still, memory access remains a bottleneck. Generic optimization techniques yield only small improvements in the memory access behavior of event-driven servers, as such techniques do not exploit their specific structure and behavior.

This paper presents an optimization framework dedicated to event-driven servers, based on a strategy to eliminate data-cache misses. We propose a novel memory manager combined with a tailored scheduling strategy to restrict the working data set of the program to a memory region mapped directly into the data cache. Our approach exploits the flexible scheduling and deterministic execution of event-driven servers.

We have applied our framework to industry-standard web servers including TUX and thttpd, as well as to the Squid proxy server and the Cactus QoS framework. Testing TUX and thttpd using a standard HTTP benchmark tool shows that our optimizations applied to the TUX web server reduce L2 data cache misses under heavy load by up to 75% and increase the throughput of the server by up to 38%.

1. Introduction

High-performance network applications have exceptional scalability requirements, as they need to keep pace with ever-increasing network speeds and user demands. For this reason, they have been studied as a separate class of applications and have been optimized extensively [2, 7, 11, 17, 18, 22, 24]. One optimization that has proved very successful in providing high efficiency in the face of heavy loads is the use of the event-driven paradigm [1, 18]. Indeed, the commercially-available servers that are recorded to deliver the highest performance [9], such as TUX [15], Flash [18] and Zeus [16] for the HTTP protocol and SER [12] for telephony, are event-driven.

The event-driven paradigm implements the processing of a task as a finite state machine, in which the transitions between machine states are triggered by events. This paradigm generalizes naturally to concurrent tasks, by implementing each task as a continuation that records its own data and machine state. As compared to process and thread-based programs that rely on the OS for scheduling, an event-based program performs its own task management, permitting the use of specialized data structures and scheduling strategies. Furthermore, in the event-driven paradigm, the concurrency between tasks is controlled, as tasks can only be interrupted at event handler boundaries, making it easy to reason about the code within an event handler, as compared to the case of process and thread-based programs, where switching between tasks can occur at any point. Finally, as compared to general event-driven programs such as GUIs, event-driven servers have a highly deterministic execution, in which one event handler typically sets up the next. This property makes it possible to reason about task behavior across a sequence of events.

Today, the conditions affecting the performance of servers are rapidly changing. Network speeds are increasing and main memory is becoming large enough to replace disks as the principal storage unit for server data. Accordingly, network transfer and disk I/O have become less of a performance bottleneck, exposing other overheads as the dominating factor in server performance. The most prominent of these overheads is that of memory accesses. Indeed, memory latency exceeds that of computations by up to two orders of magnitude. Heavy loads on the server cause an explosion in the working data set of a server, reducing the effectiveness of the data cache and thus causing a sharp degradation in performance. Generic compiler optimizations have proved ineffectual in significantly improving this memory access behavior. Nevertheless, we observe that the specific properties of event-driven network servers introduce several optimization opportunities.

This paper presents an optimization approach based on the co-design of a memory manager with a scheduler. In effect, this co-design introduces cache-awareness into the scheduling algorithm. We exploit the determinism of event-driven servers to allow the memory allocator to be able to statically predict memory requirements across a sequence of events. We exploit the fact that an event-driven server manages its own task scheduling to augment the provided scheduling algorithm to take cache behavior into account. Our approach is the most effective on programs for which the run-time footprint of a single task is guaranteed to be smaller than the L2 cache size, as is the case in many high-performance network servers. In this case, we take measures to prevent the working data set of the program from overflowing this cache.

Our optimizations are integrated into a server program through static analysis and transformation of its implementation. We provide tools that automatically carry out these operations in an event-driven C program that conforms to a memory allocation and scheduling interface specified in this work. Legacy event-driven programs can be modified to expose this interface using specific code annotations or by implementing stub functions corresponding to those in our interface. The integration process then consists of four steps. First, static analysis is used to identify the server's memory-usage behavior. Second, a customized memory allocator is generated according to the size distributions and lifetimes of the data, identified in the first step. Third, invocations of the original memory allocator in the program are replaced by invocations of the customized one. Finally, the scheduler is modified to use feedback from the customized allocator to ensure that the total data set stays in a cache-aligned, cache-sized region.

The contributions of this paper are as follows:

- We present a novel cache-aware memory allocator, called the "Stingy Allocator". This allocator can be used with any program for which the size distribution of data objects is either defined statically or can be predicted by profiling. In coordination with specific scheduling refinements, the Stingy Allocator can be used to nearly eliminate cache misses from an event-driven server.
- We present a strategy to extend the scheduler of an event-driven program to incorporate an additional cache criterion. This strategy is enabled by the Stingy Allocator.
- We describe a set of program analysis tools that apply the approach to existing programs.
- We present an evaluation of our work in the context of the following event-driven servers: The TUX, thttpd and Flash web servers, the Squid proxy server and the Cactus QoS framework.

  Specifically, we evaluate two aspects of our work: (i) The applicability of our static analysis tools and (ii) The impact of our optimizations on program throughput. For the latter, experiments conducted on TUX and thttpd show that our optimizations reduce the number of L2-cache misses by up to 75% and increase server throughput on the network by up to 38%.

The rest of the paper is organized as follows. In Section 2 we describe the architecture of event-driven programs from the point of view of caching behavior. In Section 3 we propose a novel memory allocator, the Stingy Allocator, and a scheduling strategy to prevent the working set of an event-driven program from overflowing the data cache. Section 4 describes our toolkit for applying our optimization approach. Section 5 evaluates our approach on a number of event-driven servers. Finally, Section 6 presents related work and Section 7 concludes.

2. Event-driven Servers

Event-driven servers have a rigidly defined structure, making them amenable to specific optimizations. This section gives an overview of event-driven servers, emphasizing their characteristic structure. It also discusses their cache behavior, bringing out specific caching inefficiencies and their impact on the server's performance.

2.1 Overview

An event-driven program typically consists of a single thread that loops continuously, processing a stream of events. Events may be generated on the occurrence of some I/O, or issued explicitly by the program itself. Once intercepted, an event is interpreted, and the tasks corresponding to it are considered for scheduling. Scheduling a task amounts to executing the handler associated with the event in the task's current context. Once initiated, a handler runs uninterruptedly until completion. [1]

Concurrency is managed in an event-driven program by representing each task as a continuation consisting of the current task state and a pointer to the code to be executed in response to the next event. This representation constitutes one of the biggest differences between thread and process-based servers and event-driven servers. While thread and process-based servers abstract task state as OS-level threads or processes, event-driven servers store this state in concise application-specific data structures, and are free to use and manipulate them as required.

[1] Various strategies such as event-coloring [25] and per-stage thread pools [24] have been explored as a means to scale event-driven programs to multiprocessors. We are currently in the process of exploring the extension of our work in these directions. In its current form, we assume a uniprocessor system.

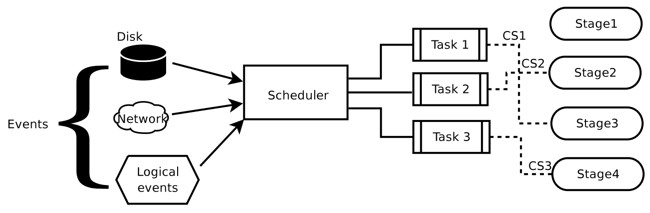


Figure 1. An event-driven server.

Event-driven servers are distinguished from other event-driven servers such as GUIs by their highly deterministic behavior. Typically, an event-driven server receives a fixed sequence of events in the processing of a given request. For example, a HTTP server first receives a request, then parses it, then processes it, *etc.* Accordingly, we can view the structure of an event-driven server as a series of stages, as illustrated in Figure 1.

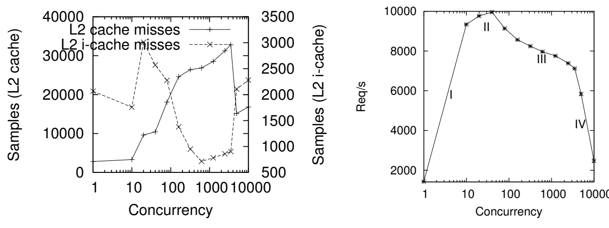
Overall, the implementation of an event-driven server can be characterized by the following elements:

- Stages: A stage is represented by a function and is bound to one or more events. It has a small number of possible predecessor and successor stages. A stage may allocate data for local use, and may allocate or use data that persists over multiple stages. Before terminating, a stage queues zero or more successor stages to be executed next.
- Tasks: A task represents the complete processing of a request. As such, it defines an execution context for the server. This execution context includes data that is shared by multiple stages of the task execution.
- Events: An event triggers the activation of a stage in a task context. Events may be *external* and generated by I/O operations or *internal* and generated by the program.
- Scheduler: The scheduler is the part of the server implementation that iteratively extracts stages waiting to be executed and executes them in their corresponding task contexts. It is typically implemented by a designated function.

2.2 Performance of Event-driven Servers

When the amount of data manipulated by a server is more than the size of the main memory, its throughput is limited by I/O activity such as disk reads. The behavior of servers under such circumstances has been widely studied [18, 10]. When the amount of memory available is sufficient to maintain this data, as is more often the case today, I/O is no longer a bottleneck. Then, the efficiency of the server implementation plays a crucial role in the performance of the server. Two aspects of the server implementation dominate its resulting performance: its interaction with the OS and its behavior with respect to the underlying hardware caches.

The event-driven architecture has been shown to be highly successful in optimizing the OS-interaction aspect of servers [1] by eliminating the need to use threads and processes to abstract tasks



(a) L2 cache misses (b) Throughput

Figure 3. Throughput degradation with increasing L2 cache misses.

altogether, and facilitating the use of efficient OS primitives for non-blocking operations. Once the bottlenecks associated with the scalability of OS primitives have been removed, the overheads associated with memory accesses become more important and can be observed to cause a significant degradation in server performance.

In the following subsection, we will study the cache behavior of event-driven servers on a highly efficient implementation of the event-driven architecture, namely, the TUX web server. Through the TUX web server, we will study the influence of the data cache on server performance.

2.3 Caching behavior

At any given time, the memory that is used by an event-driven server consists of the data that is (i) live in the contexts of the various concurrent tasks, (ii) the global state of the server, and (iii) the local data of the currently executing stage. When the total size of this data exceeds the capacity of the cache, cache misses occur, resulting in expensive main-memory accesses. The capacity of the cache depends on both its size and its associativity, *i.e.*, how many cache lines are available to represent a given memory location.

We illustrate the cache behavior of an event-driven server using the program shown in Figure 2(a). This program consists of five stages, each annotated with the objects live in the stage. For objects that are dynamically allocated, the beginning of its lifetime is marked by an explicit memory allocation and the end by an explicit deallocation. For statically allocated objects, the lifetime can be seen as the time between the first and last use of the object. In this program, we assume that all objects are the same size, and that the scheduling is round-robin.

We consider the concurrent execution of four tasks, whose processing begins simultaneously, as illustrated in Figure 2(b). Each task allocates an O2 object in its first stage. As this object is also used by stages 2 and 3, it becomes part of the context of each task. Thus, at the end of the processing of the first stage the memory requirement of the server consists of four instances of O2. Since the total size of these instances is much smaller than the size of the cache, the data can be expected to fit within it. The second and third stages allocate and then use the additional object O1. Thus, between the second and third stages, the memory requirement of the server consists of four instances of O2 as well as four instances of O1. This requirement exceeds the cache capacity. In this case, the state of the server can no longer be maintained in the cache and must be spilled into memory, requiring expensive memory accesses.

When a server is heavily loaded, memory traffic is high, and cache behavior can degrade quickly. On a cache miss, an arbitrary data item is evicted. If this item is live, which it is likely to be when the server is under heavy load, it will soon be reloaded, probably evicting another live data item, leading to a domino effect. This degradation in performance can be observed in practice. Figure 3(a) shows the change in the L2 data-cache misses when running TUX as a function of the concurrency of the workload (*i.e.*, the number of concurrent requests in flight over the network).

The performance regime in Figure 3(b) consists of three regions. In the leftmost region (marked 'I'), throughput increases constantly as the latency of packets over the network gives the server enough time to process the small batches of requests sent. Thus, increasing the number of concurrent requests uses up an increasingly large fraction of this available latency. When the concurrency increases to an extent that fully utilizes this latency, then the server begins to process multiple requests concurrently (region 'II'). We believe that the improvement in this region comes as a result of improved instruction locality. The decreasing number of i-cache misses in Figure 3(a) support this belief. In the third region (region 'III'), we find that the amount of data corresponding to the requests treated concurrently no longer fits in the L2 cache. Thus, there is a steady increase in the number of L2-cache misses along with a steady degradation in performance. Finally, in the fourth region, the server is overwhelmed with requests and spends the majority of the CPU cycles available to it in dealing with new requests. As a result, its throughput drops abruptly, and it fails.

3. Eliminating Data-Cache Misses

Our goal is to eliminate cache misses in the largest cache present on the system, *i.e.*, an external cache (e-cache), or L2 cache in the absence of an e-cache. We find this overhead to be the biggest bottleneck, and penalties arising due to L1 cache misses less significant. Indeed, on most modern processors, the difference between memory latency and L2 (or e-cache) latency is more than two orders of magnitude greater than the corresponding difference between L2 and L1 cache latencies.

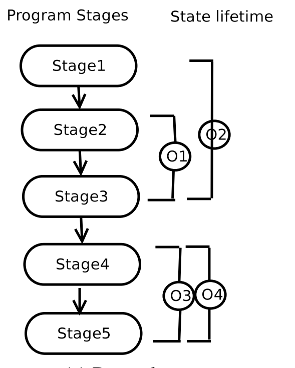
The working data set of an event-driven server comprises a stack, assumed to be allocated statically, global data, which may be allocated dynamically or statically, and per-task state, which is all data that is maintained within a stage or across multiple stages. The per-task state typically makes up the bulk of an event-driven server's working set, and is thus the main target of our optimization strategy.

In this section, we give an overview of our optimization strategy in three parts. First, we briefly describe the three main data regions that constitute the program working set. Second, we give an overview of our cache-aware memory allocator, the Stingy Allocator. Finally, we discuss the role of the scheduler in cache utilization, focusing on adjustments in the scheduler to improve cache behavior.

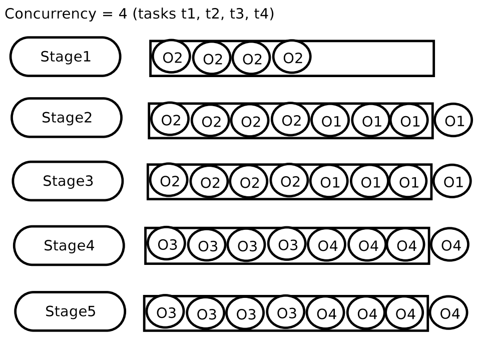
3.1 The Stingy Allocator

The Stingy Allocator is the basis of our optimization strategy to improve the cache behavior of a program. It controls where and how much memory is allocated to ensure that the data items in the working data set of a program will not cause collisions in the L2 cache. The control over where memory is allocated is obtained by allocating memory from a memory region that is mapped directly to the L2 cache[2] The control over how much memory is allocated is obtained by first analyzing the program to determine its memory requirements and then laying out this required memory in the cache-mapped region such that there are no cache collisions. All the components of the server's working data set are contained in this memory region. Furthermore, each object is assigned an area

[2] In our implementation for the Pentium II and Pentium III processors, aligning the memory region with the cache amounts to reserving and using a physically contiguous range of memory of the size of the L2 cache. The Linux 2.6 kernel provides a set of interfaces to obtain virtual memory ranges that are contiguous in physical memory.



(a) Per-task state



(b) Cache overflow

Figure 2. Per-task state during program execution

in this region and a limit is imposed on the number of instances of each object that are active at a time.

The Stingy Allocator manages a fixed number of each kind of object, and guarantees that as long as a program uses only these objects, they will not interfere with one another in the cache. The Stingy Allocator must thus be configured by selecting the number of each kind of object. This selection takes into account constraints on the size of the cache, the set of objects used within a given stage, the sequence in which the object used by a given are allocated, and the desirability of concurrency at the various stages.

The maximum memory usage that a single stage can entail is the case where one task is executing in the stage and all of the others are waiting to enter the stage. In this case, we must ensure that all of the objects live in these tasks fit within the cache. For each stage, we thus obtain the following constraint, where $\mathcal{L}$ is the set of objects live at the beginning of the stage, $\mathcal{A}$ is the set of objects allocated during the stage, $size(O_i)$ is the size of object O_i, n_{O_i} is the number of instances of object O_i managed by the Stingy Allocator, and τ is the amount of cache space allocated to per-task state:

$$\Sigma_{O_l \in \mathcal{L}}(size(O_l) \cdot n_{O_l}) + \Sigma_{O_a \in \mathcal{A}} size(O_a) \leq \tau$$

For example, in the case of stage 2 in Figure 2(a), we obtain the following constraint:

$$size(O_2) \cdot n_{O_2} + size(O_1) \leq \tau$$

We obtain further constraints from the allocation order of objects that are live in the same stages. Consider objects O2 and O1 in Figure 2(a), which are both live in stages 2 and 3. As the object O2 is allocated before the object O1, any task that is holding an O1 object must be holding an O2 object as well. Thus:

$$n_{O_2} \geq n_{O_1}$$

More generally, the relation between any two objects can be characterized in terms of the standard compiler dominance relation between their allocation and deallocation sites. That is, for any objects O_i and O_j, if the allocation site of O_i dominates that of O_j and the allocation site of O_j dominates the *deallocation* site of O_i, then we obtain the constraint:

$$n_{O_i} \geq n_{O_j}$$

The above constraints generally leave substantial latitude in the numbers of the various objects. As the number of objects managed by the Stingy Allocator determines the number of tasks that can be executing at a given stage, it is desirable to solve the constraints with respect to an objective function that maximizes the number of objects available for stages where high concurrency is beneficial, *e.g.* to improve the i-cache behavior. A detailed discussion of how to encode concurrency strategies in the objective function is out of the scope of this paper. However, for the sake of clarity, we take up an example.

Let us consider a simple model of an event-driven program's use of the instruction cache. We define a *run* of a stage as a series of consecutive executions of it for a batch of tasks. We assume that the first iteration of a run causes instructions of the stage to be fetched from the main memory so that the following iterations retrieve these instructions from the cache. Then, the cost of the first iteration of a stage is significantly greater than that of the second and subsequent iterations. Let the cost of the first iteration be defined for each individual stage i, by the quantity w_i.

We define the *instruction-fetch work* done in processing N tasks as the sum of the cost of processing the first iteration of every run involved in treating the tasks. Thus, minimizing the amount of instruction-fetch work done also minimizes the number of instruction cache misses.

If M_w is the instruction-fetch work function, S is the set of stages and $\mathcal{L}_s$ is the set of objects live in stage s, then M_w is defined as follows:

$$M_w(N) = \Sigma_{s \in S} \frac{w_s \cdot N}{\min_{O_l \in \mathcal{L}_s} n_{O_l}}$$

The intuition behind taking the minimum of the number of the objects that are live in a stage is the fact that the flow of tasks through a stage is limited by the minimum number of objects of a given kind available at the stage.

By combining the constraints dictated by the data-cache with this objective function, we obtain an *Integer Programming* minimization problem. Thus, the configuration of the Stingy Allocator, *i.e.*, the number of objects of each kind, is obtained by solving this problem.

3.2 Scheduling for cache efficiency

The Stingy allocator never allocates an object that would cause it to exceed its configured bounds. To avoid the complexity and inefficiency of starting to execute a stage only to have its memory allocation fail, we augment the server's scheduler to make it aware of the memory requirements of each stage and the current ability of the Stingy Allocator to satisfy these requirements. This information is provided by a table that maps a stage to the set of objects that are allocated by the stage and the number of those objects currently available. As an example, consider one possible solution for the number of objects O1 and O2 in Figure 2(a), $n_{O1} = 3, n_{O2} = 4$.

TUX:

```
void add_tux_atom (tux_req_t *req, atom_func_t *atom)
    __attribute__ ((QueueStage ("T","S")));
void *tux_malloc (int size)
    __attribute__ ((Malloc ("int")));
void kfree (void *mem)
    __attribute__ ((Free ("O"));
static int event_loop (threadinfo_t *ti)
    __attribute__ ((Scheduler));

tux_req_t *tux_malloc_req ();
tux_req_t *tux_malloc_req_wrap (int size)
    __attribute__ ((Malloc ("int")));
tux_req_t *tux_malloc_req_wrap (int size)
  return (tux_malloc_req());
```

Figure 4. Example annotations and wrappers for TUX.

In this case, when the scheduler is about to schedule task 4 at stage 2, it will discover that the Stingy Allocator does not have any more instances of object O2 left to allocate, and will thus select a task that is in another stage. In addition to the benefit of ensuring that an elected task can run its current stage to completion, this approach allows the scheduler to group homogeneous tasks into batches and check the availability of memory for an entire batch at a time.

4. Integrating the Stingy Allocator with legacy code

Our optimization framework consists of a set of analysis and transformation tools that are used to convert a program to use the Stingy Allocator and the associated scheduling strategy. In this section, we first describe the use of these tools by a programmer who wants to optimize an event-driven server, and then present the analyses and transformations underlying the tools.

4.1 A programmer's eye view

We take as a starting point an event-driven server written in C. Our framework requires that the server conform to a fixed interface, describing the signatures of relevant functions such as queuing a stage and allocating memory. If the program conforms to a compatible interface, as is mostly the case for TUX, Flash and Squid, source-code annotations can be used to identify the corresponding functions. If not, wrapper functions need to be introduced.

Figure 4 contains examples of the annotations and wrappers used in TUX. The `add_tux_atom` function is identified as the interface construct *QueueStage*. Its first argument is labeled with "T", indicating that it represents the task context, and the second with "S", indicating that it represents the stage to be queued. Similarly, the functions `tux_malloc` and `tux_free` are identified as the *Malloc* and *Free* constructs respectively. The `tux_malloc_req` function, used to allocate a request data structure, cannot be labeled directly as it does not accept any argument corresponding to the size of the allocated data. This function must hence be wrapped in a new function that accepts as argument the object size. Invocations of `tux_malloc_req` in the source code must then be textually replaced by invocations of `tux_malloc_req_wrap`.

Once the server has been made to conform to our interface, it can be analyzed and transformed by our tools. This process entails the following steps:

Analyzing memory utilization. The first step is to analyze the memory utilization of the program. For this purpose, we provide the tool `memwalk`, which analyzes the program and provides conservative approximations of three quantities: (i) The amount of stack used by the program (ii) The amount of per-task state allocated and deallocated categorized for the various objects. (iii) The amount of global state used by the program. The output of this tool is used in the subsequent steps.

Parameterizing the Stingy Allocator. The second step is to generate a configuration of the Stingy Allocator that corresponds to the output of `memwalk`. This output is fed into a tool named `stingygen` that yields a memory map. This memory map is to be compiled with the server implementation and is referenced by the Stingy Allocator, which is linked in as a library.

Using the allocator. The third step is to transform the program to use the Stingy Allocator in place of the original memory allocator used by the program. This step involves simply replacing the occurrences of *Malloc* in the stages, by invocations of the Stingy Allocator memory allocation function `StingyAlloc`. `StingyAlloc` differs from *Malloc* in that its argument is an index indicating the allocation site rather than the requested memory size.

Modifying the scheduler. The final step is to modify the scheduler. In this step, the programmer modifies the scheduler to incorporate the cache criterion. Figure 6 shows a round-robin scheduler for an event-driven program, before and after being modified to include this criterion. Apart from the usual scheduling criteria summarized by the function *Elect_And_Dequeue_Task*, an invocation to the `StingyDynCheck` function, defined by the Stingy Allocator library, checks whether enough instances of all the objects required to schedule a task are available.

The modification of the scheduler is the only step for which we do not provide an automatic tool, because the scheduling code may vary widely. In our experience with a variety of legacy event-driven servers (see Section 5), it is easy for the programmer to identify the code implementing the scheduling criteria and to augment this code with the appropriate use of `StingyDynCheck`.

4.2 Analyses and transformations

We now describe the analyses and transformations implemented in the `memwalk` and `stingify` tools.

4.2.1 Identifying the stages and the scheduler

To analyze the stack, global, and per-task state of a program, `memwalk` needs to identify the stages and to distinguish the code implementing these stages from the implementation of the scheduler. We begin with the identification of the stages. We represent the program stages using a graph called the *Stage Call Graph (SCG)*. Nodes in the SCG represent functions, and edges represent either a call relationship (function A calls function B) or a queuing relationship (function A queues function B to be scheduled). Call relationships are referred to as as *call edges*, and queuing relationships as *event edges*. Call edges are indicated by C language function calls. Event edges are indicated by analyzing invocations of the construct *Queue_Stage*, described in Figure 5. Thus, if a function A invokes $Queue\_Stage(atask, B)$ then an event edge is added between the nodes corresponding to A and B. In either case, the destination of the edge may be represented by a function pointer. Thus, our implementation provides an alias analysis that enumerates all the aliases of the function value.

Once the SCG has been built, the stages are the call-edge connected components that are reachable from at least one event edge. Sometimes, an edge has both incoming call edges and incoming event edges. We treat such cases by duplicating the corresponding node in the graph, so that the call-edge pointed copy becomes part of a larger stage, and the event-edge-pointed copy becomes the entry point of another stage. Sometimes, the constructed SCG can consist of many connected components, corresponding to independent functionalities of the server (such as implementations of

Queue_Stage : $S \times T \rightarrow void$	A function that queues a task to be executed at a particular stage.
Scheduler : $void \rightarrow void$	The implementation of the scheduler.
Malloc : $int \rightarrow O$	A function to allocate a block of memory for an object in O.
Free : $O \rightarrow void$	A function that frees the memory allocated for an object in O.
Where,	
$S \subset [0, \infty)$	is the set of stages.
$T \subset [0, \infty)$	is the set of tasks.
O	is the set of objects used by various stages in the course of processing tasks. Each allocation site corresponds to a distinct object.

Figure 5. Interface used to extract the structure and memory utilization behavior.

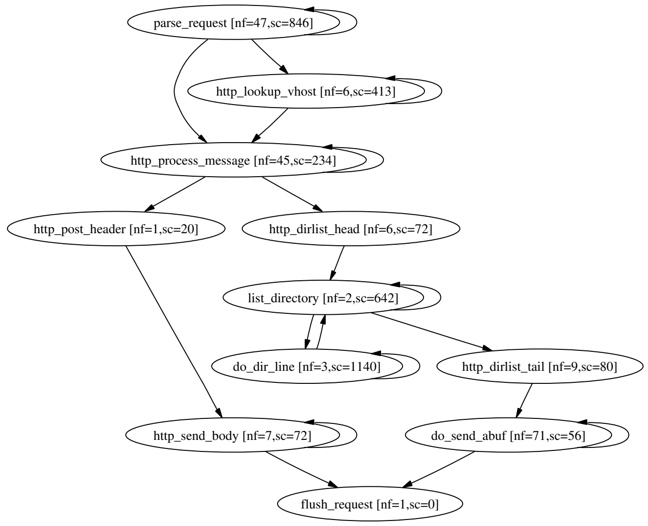
```
while (1) {                                                  // as long as the program is running
    while (workqueue.events_pending) {                       // the workqueue is not empty
      cur_task = Elect_And_Dequeue_Task(workqueue);
      Schedule_Stage(cur_task.stage, cur_task.context);
    }
    Sleep_And_Wait_For_Events();
}
```

(a) Original

```
while (1) {                                                  // as long as the program is running
    while (workqueue.events_pending) {                       // the workqueue is not empty
      cur_task = Elect_And_Dequeue_Task(workqueue,           // StingyDynCheck is now passed
                                        StingyDynCheck);     // as a predicate.
      Schedule_Stage(cur_task.stage, cur_task.context);
    }
    Sleep_And_Wait_For_Events();
}
```

(b) Modified

Figure 6. Modifying the scheduler



parse_request [nf=47,sc=846]
http_lookup_vhost [nf=6,sc=413]
http_process_message [nf=45,sc=234]
http_post_header [nf=1,sc=20]
http_dirlist_head [nf=6,sc=72]
list_directory [nf=2,sc=642]
do_dir_line [nf=3,sc=1140]
http_dirlist_tail [nf=9,sc=80]
http_send_body [nf=7,sc=72]
do_send_abuf [nf=71,sc=56]
flush_request [nf=1,sc=0]

Figure 7. A fragment of the Stage Call Graph (SCG) of TUX. *nf* denotes the number of functions in a stage, and *sc* its maximum stack utilization in bytes. Functions belonging to the same stage have been collapsed into a node representing the stage.

different protocols). Some of these functionalities may include undesired ones that pollute the SCG. For example, the Squid proxy server uses its event interfaces for interacting with users, application timeouts *etc.*. As such functionalities are likely not subject to heavy loads, they need not be optimized using the Stingy Allocator. The exploration of an SCG may be started at a particular node specified as input to the `memwalk` tool.

The entry point of the scheduler is the function implementing the *Schedule* item of the interface. The analysis identifies as the scheduler all of the functions reachable by a depth-first traversal from the node representing this function up to the entry points of the stages

A part of the SCG obtained by analyzing the TUX Web server is summarized in Figure 7. To save space, we have collapsed functions belonging to a particular stage into a node representing that stage. Each node in this modified SCG is annotated with the total number of functions that constitute the corresponding stage, along with its estimated stack utilization, as calculated by the analysis in the following section.

4.2.2 Memory Analysis

The second analysis carried out by `memwalk` is the analysis of memory used by the stages. We will describe this separately for each type of state.

Analysis of per-task state The analysis of per-task state can be seen as an analogue of the liveness analysis performed by an optimizing compiler. An object is live between the time that it is allocated and the time that it is deallocated. Each stage is associated with a transfer function that updates the set of live objects. When an object is allocated using the *Malloc* construct, it is added to the live set, and when one is deallocated using the *Free* construct, it is removed from the live set. At the end of the analysis, those objects that are live at terminal stages are assumed global. The remaining objects belong to per-task state.

The liveness of objects also determines object dependencies. If two objects are live in the same stage, then they are dependent and may not share memory (in much the same way as program variables that are live together may not share the same registers). Such objects are assigned to different memory regions by the Stingy Allocator. Objects that cannot be allocated simultaneously for a task may be assigned the same region. To maximize memory utilization, the dependency analysis colors a graph with objects at the nodes and edges specifying a dependency between objects. The number of colors is minimized to maximize the utilization of the cache. Each color is ultimately allocated a separate region by the Stingy Allocator.

Cycles in the SCG are identified by enumerating strongly-connected components using Tarjan's algorithm [21]. For objects allocated in such cycles, the number of instances that may be allocated per task is unbounded. Thus, by default, these objects are not managed by the Stingy Allocator and continue to use the original memory allocator. Alternatively, such data can be managed by the Stingy Allocator if the programmer annotates the code with an estimate of the number of instances of each such object that may be allocated per task. The occurrence of cycles in the SCG usually corresponds to situations in which a stage queues itself to be executed in the future due to the unavailability of input data, or a temporary failure. For example, the *parse_request* stage in 7 queues itself when the input buffer received by it is incomplete, and hence cannot be parsed in the current iteration. In most such situations, objects that are live in the stage are usually allocated in the first iteration and preserved over subsequent ones. Indeed, loops in the SCG in which every iteration performs an allocation are rare.

Analysis of global state Global state is identified during the analysis of per-task state, as described above. Once an object has been classified as global, it is sub-classified based on its size. Objects smaller than a user-defined threshold are labeled *final* and those that are larger are labeled *temporary*. Final objects are kept permanently in the cache, while temporary objects are stored in uncached memory and copied in on demand.[3] The threshold used for the classification typically depends on the object size distribution and the size of the cache. Keeping too many or too large objects in the cache permanently can leave insufficient room for per-task state, reducing the number of tasks that can be treated concurrently.

Stack analysis The total stack space required is calculated to be the sum of the maximum stack used by the scheduler before scheduling a task, added to the maximum stack used by the stages. The amount of stack required for a function is computed as the sum of two quantities: (i) The amount of memory consumed by all the local variables and (ii) The stack required to call the function, which includes the sizes of the arguments, the return address and saved registers. The amount of stack required along a path is calculated by summing the stack requirement for each function along the path. Recursion is handled in the same way as cycles in the SCG, by identifying strongly-connected components in the call graph.

[3] An explanation of the implementation of this aspect of the Stingy Allocator is omitted for brevity.

If the stack usage varies widely, then allocating the maximum amount required may result in many locations in the cache-aligned region that are very rarely used. Indeed, exceeding the cache region allocated to the stack does not result in a program crash due to a stack overflow, but causes the stack to spill out of the Stingy Allocator's cache-aligned region resulting in cache misses. Although this situation is undesirable, it does not prevent the program from functioning correctly. Thus, the `memwalk` tool not only calculates the maximum size of the stack, but also summarizes the sizes around which the stack utilization of various paths is clustered. The programmer may edit this information before passing it to `stingygen`, to choose a smaller stack size if desired.

4.2.3 Parameterizing the Stingy Allocator

Using the memory utilization information provided by `memwalk`, the tool `stingygen` generates a configuration for the Stingy Allocator, which includes a memory map and some data structures used for accounting. The memory map is based on an analysis of the sizes of different state components. For the stack and final global state, a fixed amount of space is set aside permanently, while for the per-task state, dependencies between objects as discussed in Section 3 are used to distribute objects into different regions. The sizes of the individual regions depend on the calculated values of the per-object limits, as calculated in Section 3.

4.2.4 Code Annotations

The Stingy Allocator relies on determinism in the memory utilization behavior of the program. However, in the presence of features such as recursion and dynamically sized buffers, statically determining the memory utilization is not possible. To enable the optimization of programs in the presence of these features, we propose the use of specific annotations in the source code. These annotations are currently provided as C language attributes. These annotations have already been mentioned in Section 4.1.

Figure 8 illustrates such an annotation in the source code of a server, and a sugare version of the output generated by the `stingify` tool. The new attributes we introduce, `stingy_size` and `stingy_count` can be used to provide an optimistic estimate of the size of a dynamically sized object, or that of the maximum number of per-task instances of an object in the case of dynamic loops, recursion and SCG cycles. The code generated for the allocation of the object contains a guard to check if the specified size of the object is exceeded, and accordingly uses the default allocator or the Stingy Allocator. No such guard is required in the corresponding deallocation of the object, as objects allocated by the Stingy Allocator can be identified on the basis of their virtual memory addresses.

The estimates passed to the tools may be intuitive, or obtained by profiling. As an example of the former, TUX contains a cycle in its SCG at the request parsing stage. This loop ensures that parsing begins only when a request has been fully received. Although the analysis reports that the request buffer allocated in this stage has potentially unbounded instances, examining the code reveals that this buffer is only allocated on the first iteration. Thus, an annotation is added to set its `stingy_count` attribute to 1. Although generating these estimates is error-prone, the use of a guard guarantees that a misestimate does not corrupt the functioning of the program. At worst, providing a wrong estimate reduces performance.

5. Experimental evaluation

We have applied our tools to five event-driven programs: The TUX, thttpd and Flash web servers, a test server using the Cactus QoS framework and the Squid proxy server. In the first part of this section, we discuss the applicability of our approach by giving an

Input:

```
#define STINGY_DIR_SIZE 148
char *dir_name __attribute__ ((stingy_size (STINGY_DIR_SIZE));
dir_name = (char *) Malloc(strlen(request->well_formed_url);
```

Output:

```
char *dir_name;
int __tmp0 = strlen(request->well_formed_url);
if (__tmp0 < 148) {
  dir_name = StingyAlloc(ID_DIR_NAME);
}
else {
  dir_name = (char *) Malloc(strlen(request->well_formed_url);
}
```

Figure 8. Guiding the tools using code annotations.

overview of the effort involved in processing these programs. All these programs, with the exception of Flash, are available publicly. The Cactus QoS framework is distributed as a library along with the implementation of an example transport protocol (CTP). We applied our tools to a test server that uses this protocol.

To evaluate the performance benefits of our approach, we evaluated the performance of unmodified versions of TUX and thttpd on a real network using a standard benchmarking tool for HTTP servers [13], and then did the same for a version optimized using our toolkit. In Section 5.2 we present an analysis of these experiments.

5.1 Applicability

We have claimed that the interface used by our tools for the purpose of analysis and transformation is general and widely applicable. In this section, we support this claim with evidence in the form of actual code excerpts, shown in Figure 10, containing some representative wrappers and annotations written to apply our toolkit to the programs considered.

In Cactus, a stage specifies the next event to be executed using the function cRaiseEvent. This function is used by the current stage to specify the next event to be scheduled for the current task. Since a function annotated with *QueueStage* needs to accept a pointer to a stage function, `cRaiseEvent` is wrapped in a function that accepts an additional argument of a pointer to a function. The field, `lBinding->p` in the event data structure contains the function pointer that the event is bound to. As mentioned in Section 4, an alias analysis collapses this function pointer into a set of candidate successor stages. This stage is passed as an additional parameter to the function.

In thttpd, the scheduler looks up the stage to be executed in a particular context using a connection state, represented by an enumerated type. Thus, queuing the next stage to be scheduled amounts to modifying the value of the connection state. This functionality is thus wrapped in a new function, which accepts the additional parameter of the function corresponding to the stage to be executed, along the lines of the previous example.

Collapsed SCGs corresponding to Flash, the Squid server and thttpd are shown in Figure 9. Other SCGs are left out to save space.

5.2 Evaluation

We ran our experiments on a Gigabit network between a powerful client system with two Intel Xeon processors running at 3GHz, and a server system with an Intel Pentium III M running at 1.4GHz. The server has 1Mb of L2 cache and 2GB of RAM. The aim of using a powerful machine as the client is to ensure that operation is bottlenecked on the server. We verified that this was the case using

Cactus:

```
extern int cRaiseEvent( cevent *pev, ceventmode em,
        int nDelay, int nUrgency, int cDynamicArgs);
extern int cRaiseEvent_wrap( cevent *pev, funcptr_t
        *func_pointer, ceventmode em, int nDelay, int
        nUrgency, int cDynamicArgs)
        __attribute__ ((QueueStage
                          ("X","S","X","X","X","X")));
extern int cRaiseEvent_wrap( cevent *pev, funcptr_t *
        func_pointer, ceventmode em, int nDelay,
        int nUrgency, int cDynamicArgs)
{
        cRaiseEvent(pev, em, nDelay, nUrgency, cDynamicArgs);
}
```

Example invocation:

```
cRaiseEvent_wrap (pev, pev->lBinding->p,
                    em, nDelay, nUrgency, cDynamicArgs);
```

thttpd:

```
extern int QueueStage (connecttab *c, int state,
               funcptr_t *func_pointer) {
c->conn_state = state;
}
```

Example invocation:

```
QueueStage(c, CNST_READING, handle_read)
```

Flash:

```
void
SetSelectHandler(int fd, SelectHandler s, int forRW)
__attribute__ ((QueueStage ("X", "S", "X")));
```

Squid:

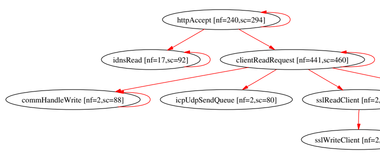
```
void eventAdd(const char *name, EVH * func,
      void *arg, double when, int weight)
      __attribute__ ((QueueStage ("X","S","X","X","X")));


void commSetSelect(int fd, unsigned int type,
      PF * handler, void *client_data, time_t timeout)
      __attribute__ ((QueueStage ("X","X","S","X","X")));
```

Figure 10. Excerpts of Code Annotations for the Programs. The test programs are annotated and instrumented with wrappers to expose a standard interface for the purpose of analysis.

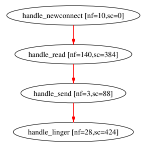
the Netperf benchmark suite [8]. We used the Apachebench [13] package to measure performance. Apachebench has been previously used to evaluate HTTP server performance in the systems community [23, 24].

Figures 11(a) and 12(a) show the variation of requests serviced per second with increasing concurrency in the two servers. Figures 11(b) and 12(b) show the corresponding variation in L2 cache misses. For the most intense loads, the requests serviced by TUX increase by about 38% and L2 cache misses decrease by about 75%. For thttpd, the throughput increases by about 12% and the number of L2 cache misses by about 53%.

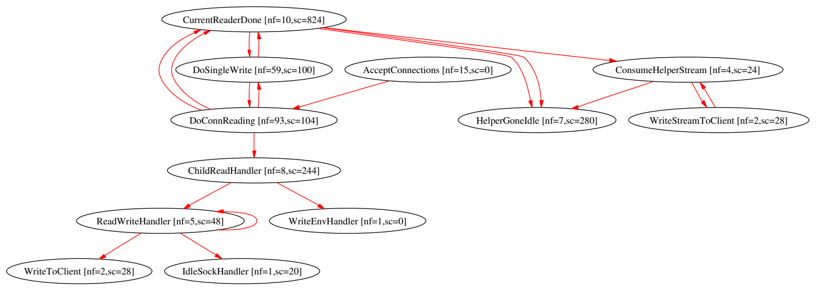
We attribute the massive difference between the improvements observed in the two servers to the difference in their original implementations. TUX is highly optimized and makes use of low-level OS interfaces to achieve the highest possible efficiency [15]. On the contrary, thttpd is an ordinary http server that uses standard OS mechanisms and is not known as a high performance server. As one may observe in Figures 11(a) and 12(a), the absolute through-


httpAccept [nf=240,sc=294]
idnsRead [nf=17,sc=92]
clientReadRequest [nf=441,sc=460]
commHandleWrite [nf=2,sc=88]
icpUdpSendQueue [nf=2,sc=80]
sslReadClient [nf=2,sc=64]
sslReadServer [nf=1,sc=64]
sslWriteClient [nf=2,sc=56]
sslWriteServer [nf=1,sc=56]

(a) Squid

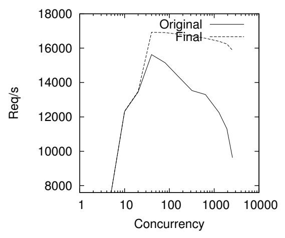

handle_newconnect [nf=10,sc=0]
handle_read [nf=140,sc=384]
handle_send [nf=3,sc=88]
handle_linger [nf=28,sc=424]

(b) thttpd

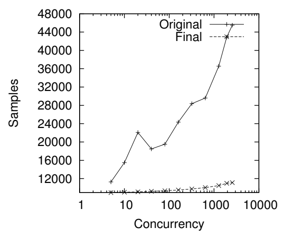

CurrentReaderDone [nf=10,sc=824]
DoSingleWrite [nf=59,sc=100]
AcceptConnections [nf=15,sc=0]
ConsumeHelperStream [nf=4,sc=24]
DoConnReading [nf=93,sc=104]
HelperGoneIdle [nf=7,sc=280]
WriteStreamToClient [nf=2,sc=28]
ChildReadHandler [nf=8,sc=244]
ReadWriteHandler [nf=5,sc=48]
WriteEnvHandler [nf=1,sc=0]
WriteToClient [nf=2,sc=28]
IdleSockHandler [nf=1,sc=20]

(c) Flash

Figure 9. Portions of collapsed SCGs for the test programs (Call-edges have been deleted.)

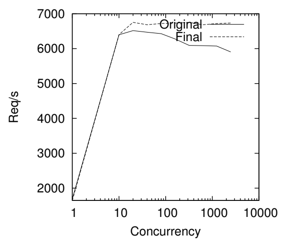


(a) Throughput

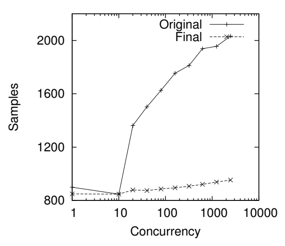


(b) L2 cache misses

Figure 11. Comparison of the performance of the original TUX server to that of the optimized TUX server



(a) Throughput



(b) L2 cache misses

Figure 12. Comparison of the performance of the original thttpd server to that of the optimized thttpd server

put of TUX is about 2.5 times that of thttpd. We consider that TUX is representative of the target applications of our work because it is already highly optimized, making the cache bottleneck all the more significant.

The cache misses that remain even after the inclusion of the Stingy Allocator occur due to interference with modules on which the servers depend that are not modified to use the Stingy Allocator. Such modules include OS modules such as the protocol stack and the file system drivers and external library functions. In order to entirely eliminate data-cache misses, one would need to include these modules in the optimization process through explicit OS support for the Stingy Allocator. We are currently in the process of exploring this extension.

6. Related Work

With the increasing gap between microprocessor speeds and memory access times, the optimization of cache usage has been an intensively researched topic in the compiler and systems community. Due to its sheer size, it is impossible to cover the full body of work in the domain. We will focus on the works that are the most related to ours.

Larus and Parkes presented *Cohort scheduling* [14], which is another scheduling strategy to improve the cache performance of concurrent programs. Cohort scheduling induces consecutive runs of stages by accumulating (cohorting) tasks at specific stages. This policy is configured by heuristics based on inter-stage delays and queue lengths, and has the most positive impact on static global state and the instruction cache. Our work strikes a balance between per-task state and global state by using static analysis. Better instruction-cache locality was also the goal of Blackwell [3], in his work on optimizing TCP/IP stacks. He showed that by processing several packets in a loop at every layer, one could induce better reuse of the corresponding instructions.

Cache-conscious data placement has been used to optimize the caching behavior of generic programs [4, 5, 6]. These works use program analysis and profiling information to efficiently arrange objects in memory, and fields within objects. While the goal of these efforts is to reduce the number of cache misses in generic programs, our work focuses on the specific problem of reducing data cache misses in concurrent programs. Specifically, although such data placement can be beneficial for a certain number of object instances, it does not address the situation in which the number of these instances is multiplied as a result of increasing concurrency.

Rajagopalan *et al.* have considered the problem of improving the performance of event-driven programs in general [19]. As this class of programs includes programs such as GUIs that depend heavily on user interaction and are thus highly non-deterministic, their approach relies on dynamic profiling to identify commonly occurring event-handler sequences rather than the static analysis used in our approach. This reliance on dynamic profiling implies that they can only optimize synchronous events, as it is only in this case that there is guaranteed to be a connection between two event handlers that occur in sequence. In contrast, our approach is independent of whether events are synchronous or asynchronous. The kinds of optimizations performed are also quite different, as they consider primarily optimizations in the call-and-return interface between event handlers such as function inlining, whereas we consider cache behavior. These optimizations are orthogonal, and applying both kinds of optimizations to servers that raise many synchronous events could yield further speedups.

Finally, the last body of related work optimizes concurrent programs (both event-driven and thread-based) through intelligent resource management. SEDA [24] and Capriccio [23] use *resource monitors* to implement resource-aware scheduling. These monitors are installed in various stages, and periodically sample the level of utilization of CPU, memory and file descriptors. The scheduler uses the values of resource consumption at these monitors to favor the scheduling of stages that free resources. Although this policy improves overall performance by de-prioritizing resource bottlenecks, it does not expressly go to improve caching behavior.

7. Conclusion and Future Work

We have presented an optimization framework to remove the dominant bottleneck of memory accesses from event-driven servers. Our framework includes a novel memory allocator that compacts the program's working set in a fixed region of memory, restricting it to the L2 cache, and a scheduling strategy that coordinates with the allocator to ensure that the scheduling of tasks in the server respects these constraints. A set of program analysis tools has been implemented to apply our optimizations.

We have applied our approach to the five test programs to evaluate the applicability of our work. Benchmarking the TUX and thttpd web servers has shown that data-cache misses are reduced by up to 75%, and the overall throughput of the server increases by up to 38%.

As future work, we would like to explore techniques to scale event-driven servers to multiprocessors [25] in the context of our optimizations. We are also working on a system to encode specific concurrency strategies as objective functions when configuring the Stingy allocator. Finally, we are working on a framework to integrate the Stingy allocator into thread and process-based servers.

References

[1] G. Banga, P. Druschel, and J. Mogul. Better operating system features for faster network servers. In *Workshop on Internet Server Performance*, June 1998.

[2] A. Begel, S. McCanne, and S. Graham. BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture. In *Proceedings of the ACM SIGCOMM'99 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 123–134, Cambridge, MA, USA, August 1999.

[3] T. Blackwell. Speeding up protocols for small messages. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 85–95, Stanford, CA, August 1996.

[4] B. Calder, C. Krintz, S. John, and T. Austin. Cache-conscious data placement. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 13–24, Atlanta, GA, October 1998.

[5] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, May 1999.

[6] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation (PLDI'99)*, pages 1–12, Atlanta, GA, May 1999.

[7] D. Clark, V. Jacobson, J. Romkey, and M. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[8] Hewlett-Packard company Information Networks Division. *Netperf: A network performance benchmark*, February 1996.

[9] Standard Performance Evaluation Corporation. The SPECWeb99 Benchmark. quarterly results. URL: http://www.spec.org/osg/web99/results/.

[10] K. Elmeleegy, A. Chanda, A. L. Cox, and W. Zwaenepoel. Lazy asynchronous i/o for event-driven servers. In *USENIX Annual Tech Conference*, June 2004.

[11] D.R. Engler and M.F. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In SIGCOMM'96 [20], pages 26–30.

[12] Fraunhofer Fokus. Sip express router. URL: www.iptel.org/ser.

[13] The Apache Foundation. Apache HTTP server project. URL: http://www.apache.org.

[14] J. R. Larus and M. Parkes. Using cohort scheduling to enhance server performance. In *USENIX Annual Tech Conference*, pages 103–114, Monterey, CA, October 2002.

[15] C. Lever, M. Eriksen, and S. Molloy. An analysis of the TUX web server. Technical Report 00-8, University of Michigan, November 2000.

[16] Zeus Technology Limited. Zeus web server. URL: www.zeus.co.uk.

[17] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–269, San Diego, CA, USA, January 1993. USENIX.

[18] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *USENIX Annual Tech Conference*, pages 199–212, Monterey, CA, June 1999.

[19] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Profile-directed optimization of event-based programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 106–116, Berlin, Germany, 2002. ACM Press.

[20] *SIGCOMM Symposium on Communications Architectures and Protocols*, Stanford University, CA, August 1996. ACM Press.

[21] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[22] R. van Renesse. Masking the overhead of protocol layering. In SIGCOMM'96 [20], pages 96–104.

[23] R. Von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 268–281, Bolton Landing (Lake George), New York, October 2003.

[24] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 230–243,

Banff, Canada, October 2001.

[25] N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazieres, and M-. F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX Annual Tech Conference*, pages 239–252, San Antonio, TX, June 2003.