USENIX Association

Proceedings of the
2002 USENIX Annual Technical
Conference

Monterey, California, USA
June 10-15, 2002

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Using Cohort Scheduling to Enhance Server Performance

James R. Larus and Michael Parkes
{larus, mparkes}@microsoft.com
Microsoft Research
One Microsoft Way
Redmond, WA 98052

Abstract

A server application is commonly organized as a collection of concurrent threads, each of which executes the code necessary to process a request. This software architecture, which causes frequent control transfers between unrelated pieces of code, decreases instruction and data locality, and consequently reduces the effectiveness of hardware mechanisms such as caches, TLBs, and branch predictors. Numerous measurements demonstrate this effect in server applications, which often utilize only a fraction of a modern processor's computational throughput.

This paper addresses this problem through *cohort scheduling*, a new policy that increases code and data locality by batching the execution of similar operations arising in different server requests. Effective implementation of the policy relies on a new programming abstraction, *staged computation*, which replaces threads. The *StagedServer* library provides an efficient implementation of cohort scheduling and staged computation. Measurements of two server applications written with this library show that cohort scheduling can improve server throughput by as much as 20%, by reducing the processor cycles per instruction by 30% and L2 cache misses by 50%.

1 Introduction

A server application is a program that manages access to a shared resource, such as a database, mail store, file system, or web site. A server receives a stream of requests, processes each, and produces a stream of results. Good server performance is important, as it determines the latency to access the resource and constrains the server's ability to handle multiple requests. Commercial servers, such as databases, have been the focus of considerable research to improve the underlying hardware, algorithms, and parallelism, as well as considerable development to improve their code.

Much of the hardware effort has concentrated on the memory hierarchy, where rapidly increasing processor speed and parallelism and slowly declining memory access time created a growing gap that is a major performance bottleneck in many systems. In recent processors, loading a word from memory can cost hundreds of cycles, during which three to four times as many instructions could execute. High performance processors attempt to alleviate this performance mismatch through numerous mechanisms, such as caches, TLBs, and branch predictors [27]. These mechanisms exploit a well-known program property—spatial and temporal reuse of code and data—to keep at hand data that is likely to be reused quickly and to predict future program behavior.

Server software often exhibits less program locality and, consequently achieves poorer performance, than other software. For example, many studies have found that commercial database systems running on-line transaction processing (OLTP) benchmarks incur high rates of cache misses and instruction stalls, which reduce processor performance to as low as a tenth of its peak potential [4, 9, 20]. Part of this problem may be attributable to database systems' code size [28], but their execution model is also responsible.

These systems are structured so that a process or thread runs for a short period before invoking a blocking operation and relinquishing control, so processors execute a succession of diverse, non-looping code segments that exhibit little locality. For example, Barroso et al. compared TPC-B, an OLTP benchmark whose threads execute an average of 25K instructions before blocking, against TPC-D, a compute-intensive decision-support system (DSS) benchmark whose threads execute an average of 1.7M instructions before blocking [9]. On an AlphaServer 4100, TPC-B had an L2 miss rate of 13.9%, an L3 miss rate of 2.7%, and overall performance of 7.0 cycles per instruction (CPI). By contrast, TPC-D had an L2 miss rate of 1.2%, an L3 miss rate of 0.32%, and a CPI of 1.62.

Instead of focusing on hardware, this paper takes an alternative—and complementary—approach of modifying a program's behavior to improve its performance. The paper presents a new, user-level software architecture that enhances instruction and data locality and increases server software performance. The architecture consists of a scheduling policy and a programming model. The policy, *cohort scheduling*, con-

secutively executes a cohort of similar computations that arise in distinct requests on a server. Computations in a cohort, because they are at roughly the same stage of processing, tend to reference similar code and data, and so consecutively executing them improves program locality and increases hardware performance. *Staged computation*, the programming model, provides a programming abstraction by which a programmer can identify and group related computations and make explicit the dependences that constrain scheduling. Staged computation, moreover, has the additional benefits of reducing concurrency overhead and the need for expensive, error-prone synchronization.

We implemented this scheduling policy and programming model in a reusable library (*StagedServer*). In two experiments, one with an I/O-intensive server and another with a compute-bound server, code using StagedServer performed significantly better than threaded versions. StagedServer lowered response time by as much as 20%, reduced cycles per instruction by 30%, and reduced L2 cache misses by more than 50%.

The paper is organized as follows. Section 2 introduces cohort scheduling and explains how it can improve program performance. Section 3 describes staged computation. Section 4 briefly describes the Staged-Server library. Section 5 contains performance measurements. Section 6 discusses related work.
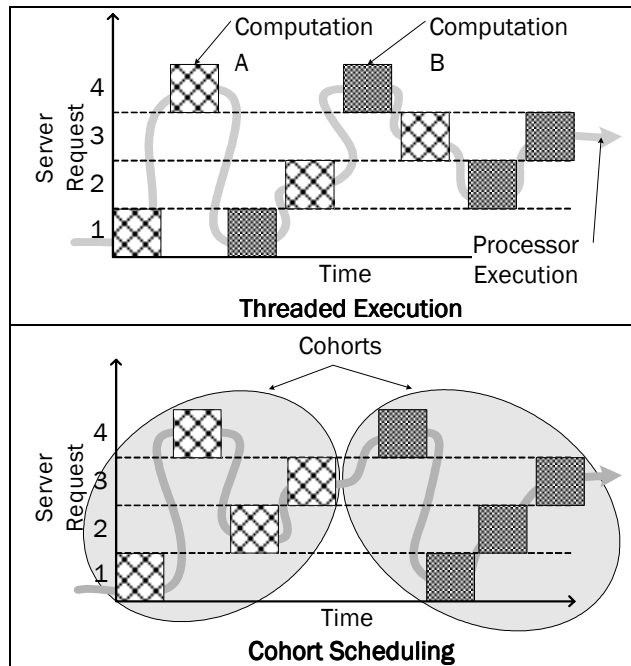

Figure 1. Cohort scheduling in operation. Shaded boxes indicate different computations performed while processing requests on a server. Cohort scheduling reorders the computations, so that similar ones execute consecutively on a processor, which increases program locality and processor performance.

2 Cohort Scheduling

Cohort scheduling is a technique for organizing the computation in server applications to improve program locality. The key insight is that distinct requests on a server execute similar computations. A server can defer processing a request until a *cohort of computations* arrive at a similar point in their processing and then execute the cohort consecutively on a processor (Figure 1).

This scheduling policy increases opportunities for code and data reuse, by reducing the interleaving of unrelated computations that causes cache conflicts and evicts live cache lines. The approach is similar to loop tiling or blocking [19], which restructures a matrix computation into submatrix computations that repeatedly reference data before turning to the next submatrix. Cohort scheduling, however, is a dynamic process that reorganizes a series of computations on items in an input stream, so that similar computations on different items execute consecutively. The technique applies to uniprocessors and multiprocessors, as both depend on program locality to achieve good performance.

Figure 2 illustrates the results of a simple experiment that demonstrates the benefit of cohort scheduling on a uniprocessor. It reports the cost, per call, of executing different sized cohorts of asynchronous writes to random blocks in a file. Each cohort ran consecutively on a system whose cache and branch table buffer had been flushed. As the cohort increased in size, the cost of each call decreased rapidly. A single call consumed 109,000 cycles, but the average cost dropped 68% for a cohort of 8 calls and 82% for a cohort of 64 calls. A direct measure of locality, L2 cache misses, also improved dramatically. With a cohort of 8 calls, L2 misses per call dropped to 17% of the initial value and further declined to 4% with a cohort of 64 calls. These improvements required no changes to the operating system code; only reordering operations in an application. Further improvement requires reductions in OS self-conflict misses (roughly 35 per system call), rather than amortizing the roughly 1500 cold start misses.

2.1 Assembling Cohorts

Cohort scheduling is not irreparably tied to staged computation, but many benefits may be lost if a programmer cannot explicitly form cohorts. For example, consider transparently integrating cohort scheduling with threads. The basic idea is simple. A modified thread scheduler identifies and groups threads with identical next program counter (nPC) values. Threads starting at the same point are likely to execute similar operations, even if their behavior eventually diverges. The scheduler runs a cohort of threads with identical nPCs before turning to the next cohort.
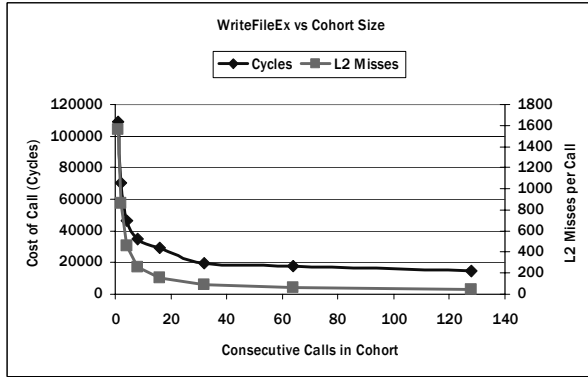
WriteFileEx vs Cohort Size

Figure 2. Performance of cohorts of WriteFileEx system calls in Window 2000 Advanced Server (Dell Precision 610 with an Intel Pentium III processor). The chart reports the cost per call—in processor cycles and L2 cache misses—of an asynchronous write to a random 4K block in a file.

It is easy to believe that this scheme could sometimes improve performance, and it requires only minor changes to a scheduler and no changes to applications. It, however, has clear shortcomings. In particular, nPC values are a coarse and indirect indicator of program behavior. Only threads with identical nPCs end up in a cohort, which misses many pieces of code with similar behavior. For example, several routines that access a data structure might belong in a cohort. Simple extensions to this scheme, such as using the distance between PCs as a measure of similarity, have little connection to logical behavior and are perturbed by compiler linking and code scheduling. Another disadvantage is that cohorts start after blocking system calls, rather than at application-appropriate points. In particular, compute-intensive applications or programs that use asynchronous I/O cannot use this scheme, as they do not block.

To correct these shortcomings and properly assemble a cohort, a programmer must delimit computations and identify the ones that belong in a cohort. Staged computation provides a programming abstraction that neatly captures both dimensions of cohorts.

3 Staged Computation

Staged computation is a programming abstraction intended to replace threads as the construct underlying concurrent or parallel programs. Stages offer compelling performance and correctness advantages and are particularly amenable to cohort scheduling. In this model, a program is constructed from a collection of *stages*, each of which consists of a group of exported operations and private data. An operation is an asynchronous procedure call, so its invocation, execution, and reply are decoupled. Moreover, a stage has *scheduling autonomy*, which enables it to control the order and concurrency with which its operations execute.

A stage is conceptually similar to a class in an object-based language, to the extent that it is a program structuring abstraction providing local state and operations. Stages, however, differ from objects in three major respects. First, operations in a stage are invoked asynchronously, so that a caller does not wait for a computation to complete, but instead continues and rendezvouses later, if necessary, to retrieve a result. Second, a stage has autonomy to control the execution of its operations. This autonomy extends to deciding when and how to execute the computations associated with invoked operations. Finally, stages are a control abstraction used to organize and process work, while objects are a data representation acted on by other entities, such as functions, threads, or stages.

A stage facilitates cohort scheduling because it provides a natural abstraction for grouping operations with similar behavior and locality and the control autonomy to implement cohort scheduling. Operations in a stage typically access local data, so that effective cohort scheduling only requires a simple scheduler that accumulates pending operations to form a cohort.

Stages provide additional programming advantages as well. Because they control their internal concurrency, they promote a programming style that reduces the need for expensive, error-prone explicit synchronization. Stages, moreover, provide the basis for specifying and verifying properties of asynchronous programs. This section briefly describes the staged programming model. Section 4 elaborates an implementation in a C++ class library.

3.1 Stage Design

Programmers group operations into a stage for a variety of reasons. The first is to regulate access to program state ("static" data) by wrapping it in an abstract data type. Operations grouped this way form an obvious cohort, as they typically have considerable instruction and data locality. Moreover, a programmer can control concurrency in a stage to reduce or eliminate synchronization for this data (Section 3.4).

The second reason is to group logically related operations to provide a well-rounded and complete programming abstraction. This reason may seem less compelling than the first, but logically related operations frequently share code and data, so collecting them in a stage identifies operations that could benefit from cohort scheduling.

The third is to encapsulate program control logic in the form of a finite-state automaton. As discussed below, a stage's asynchronous operations easily implement the reactive transitions in an event-driven state machine.
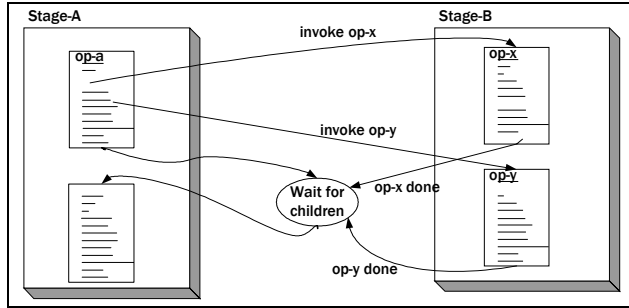
Figure 3. Example of stages and operations. Stage-A runs op-a, which invokes two operations in Stage-B and waiting until they complete before running op-a's continuation.

In practice, designing a program with stages focuses on partitioning the tasks into sub-tasks that are self-contained, have considerable code and data locality, and have logical unity. In many ways, this process is the control analogue of object-oriented design.

3.2 Operations

Operations are asynchronous computations exported by a stage. Invocation of an operation only requires its eventual execution, so the invoker and operation run independently. When an operation executes, it can invoke any number of child operations on any stage, including its own. A parent can wait for its children to finish, retrieve results from their computation, and continue processing. Figure 3 shows an operation (*op-a*) running in *Stage-A* that invokes two operations (*op-x* and *op-y*) in *Stage-B*, performs further computation, and then waits for its children. After they complete and return their results, *op-a* continues execution and processes the children's results.

The code within an operation executes sequentially and can invoke both conventional (synchronous) calls and asynchronous operations. However, once started, an operation is non-preemptible and runs until it relinquishes the processor. Programmers, unfortunately, must be careful not to invoke blocking operations that suspend the thread running operations on a processor. An operation that relinquishes the processor to wait for an event—such as asynchronous I/O, synchronization, or operation completion—instead provides a *continuation* to be invoked when the event occurs [14].

A continuation consisting of a function and enough saved state to permit the computation to resume at the point at which it suspended. *Explicit continuations* are the simplest and least costly approach, as an operation saves only its live state in a structure called a *closure*. The other alternative, *implicit continuations*, requires the system to save the executing operation's stack, so that it can be resumed. This scheme, similar to fibers, simplifies programming, at some performance cost [2].

Asynchronous operations provide low-cost parallelism, which enables a programmer to express and exploit the concurrency within an application. The overhead, in time and space, of invoking an operation is close to a procedure call, as it only entails allocating and initializing a closure and passing it to a stage. When an operation runs to completion, it does not require its own stack or an area to preserve processor state, which eliminates much of the cost of threads. Similarly, returning a value and re-enabling a continuation are simple, inexpensive operations.

3.3 Programming Styles

Staged computation supports a variety of programming styles, including software pipelining, event-driven state machines, bi-directional pipelines, and fork-join parallelism. Conceptually, at least, stages in a server are arranged as a pipeline in which requests arrive at one end and responses flow from the other. This form of computation is easily supported by representing a request as an object passed between stages. Linear pipelining of this sort is simple and efficient, because a stage retains no information on completed computations.

However, stages are not constrained to this linear style. Another common programming idiom is bi-directional pipelining, which is the asynchronous analogue of call and return. In this approach, a stage passes subtasks to one or more other stages. The parent stage eventually suspends its work on the request, turns its attention to other requests, and resumes the original computation when the subtasks produce results. This style requires that an operation be broken into a series of subcomputations, which run when results appear. With explicit continuations, a programmer partitions the computation by hand, although a compiler could easily produce this code, which is close to the well-known continuation-passing style [6, 12]. With implicit continuations, a programmer only needs to indicate where the original computation suspends and waits for the subtasks to complete.

A generalization of this style is event-driven programming, which uses a finite state automaton (FSA) to control a reactive system [26, 29]. The FSA logic is encapsulated in a stage and is driven by external events, such as network messages and I/O completions, and internal events from other asynchronous stages. An operation's closure contains the FSA state for a particular server request. The FSA changes state when a child operation completes or external events arrive. These transitions invoke computations associated with edges in the FSA. Each computation runs until it blocks and specifies the next state in the FSA.
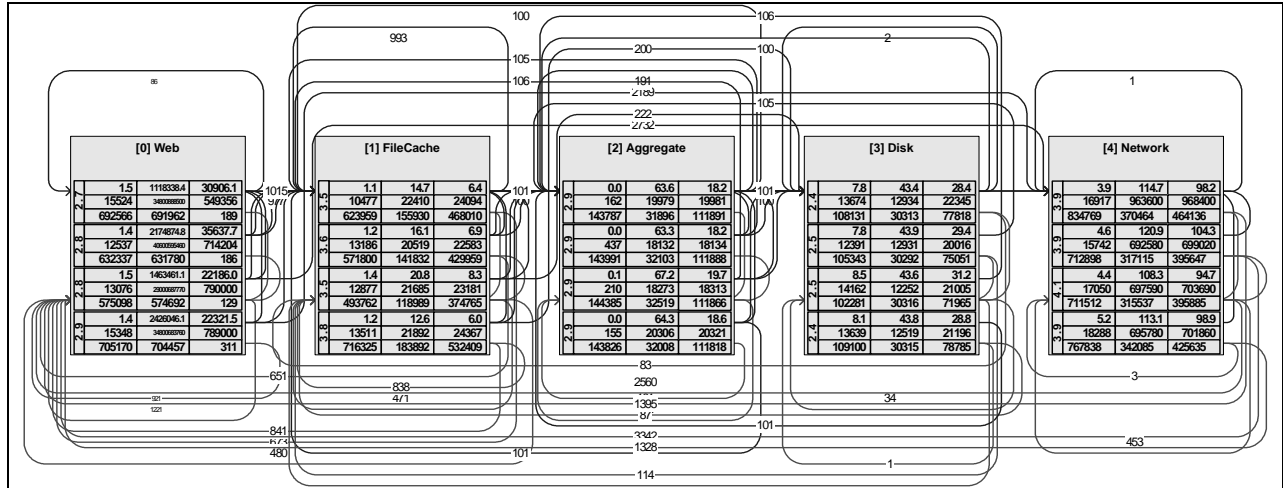
Figure 4 Profile of staged web server (Section 5.1). The performance metrics for each stage are broken down by processor (the system is running on four processors). The first column is the average queue length. The second column contains three metrics on operations at the stage: the quantity, the average wait time (millisecond), and the maximum wait time. The third column contains corresponding metrics for operations that are suspended and restarted. The fourth column contains corresponding metrics for completed operations. The numbers on arcs are the number of operations started or restarted between two stage-processor pairs.

For example, the web server used in Section 5.1 is driven by a control-logic stage consisting of a FSA with fifteen states. The FSA describes the process by which a HTTP GET request arrives and is parsed, the referenced file is found in the cache or on disk, the file blocks are read and transmitted, and the file and connection are closed.

Describing the control logic of a server as a FSA opens the possibility of verifying many properties of the entire system, such as deadlock freedom, by applying techniques, such as model checking [15, 22], developed to model and verify systems of communicating FSAs.

3.4 Scheduling Policy Refinements

The third attribute of a stage is scheduling autonomy. When a stage is activated on a processor, the stage determines which operations execute and their order. This scheduling freedom allows several refinements of cohort scheduling to reduce the need for synchronization. In particular, we found three policies useful:

- An *exclusive stage* executes at most one of its operations at a time. Since operations run sequentially and completely, access to stage-local data does not need synchronization. This type of a stage is similar to a monitor, except that its interface is asynchronous: clients delegate computation to the stage, rather than block to obtain access to a resource. When this strict serialization does not cause a performance bottleneck, this policy offers fast, error-free access to data and a simple programming model. This approach works well for fine-grained operations, as the cost of acquiring and releasing the

stage's mutex can be amortized over a cohort of operations [25].

- A *partitioned stage* divides invocations (based on a key passed as a parameter), to avoid sharing data among operations running on different processors. For example, consider a file cache stage that partitions requests using a hash function on the file number. Each processor maintains its own hash table of in-memory disk blocks. Each hash table is accessed by only one processor, which enhances locality and eliminates synchronization. This policy, which is reminiscent of shared-nothing databases, permits parallel data structures without fine-grain synchronization.

- A *shared stage* runs its operations concurrently on many processors. Since several operations in a stage can execute concurrently, shared data accesses must be synchronized.

Other policies are possible and could be easily implemented within a stage.

It is important keep in mind that these policies are implemented within the more general framework of cohort scheduling. When a stage is activated on a processor, it executes its outstanding operations, one after another. Nothing in the staged model requires cohort scheduling. Rather the programming model and scheduling policy naturally fit together. A stage groups logically related operations that share data and provides the freedom to reorder computations. Cohort scheduling exploits scheduling freedom by consecutively running similar operations.

3.5 Understanding Performance

A compelling advantage of the Staged model is that the performance of the system is relatively easy to visualize and understand. Each stage is similar to a node in a queuing system. Parameters, such as average and maximum queue length, average and maximum wait time, and average and maximum processing time, are easily measured and displayed (Figure 4). These measurements provide a good overview of system performance and help identify bottlenecks.

3.6 Stage Computation Example

As an example of staged computation, consider the file cache used by the web server in Section 5.1. A file cache is an important component in many servers. It stores recently accessed disk blocks in memory and maps a file identifier and offset to a disk block.

The staged file cache consists of three partitioned stages (Figure 5). The cache is logically partitioned across the processors, so each one manages a unique subset of the files, as determined by the hashed file identifier. Alternatively, for large files, the file identifier and offset can be hashed together, so a file's disk blocks are stripped across the table. Within the stage, each processor maintains a hash table that maps file identifiers to memory-resident disk blocks. Since a processor references only its table, accesses require no synchronization and data does not migrate between processor caches.

If a disk block is not cached in memory, the cache invokes an operation on the *I/O Aggregator* stage, whose role is to merge requests for adjacent disk blocks to improve system efficiency. This stage utilizes cohort scheduling in a different way, by accumulating I/O requests in a cohort and combining them into a larger I/O request on the operating system.

The *Disk I/O* stage reads and writes disk blocks. It issues asynchronous system calls to perform these operations and, for each, invokes an operation in the *Event Server* stage describing a pending I/O. This operation suspends until the I/O completes. This stage interfaces the operating system's asynchronous notification mechanism to the staged programming model. It utilizes a separate thread, which waits on an I/O Completion Port that the system uses to signal completion of asynchronous I/O. At each notification, this stage matches an event with a waiting closure, which it re-enables and passes the information from the Completion Port. The Disk I/O stage, in turn, returns disk blocks to the I/O Aggregator, which passes them to the FileCache stage, where the data are recorded and passed back to the client.
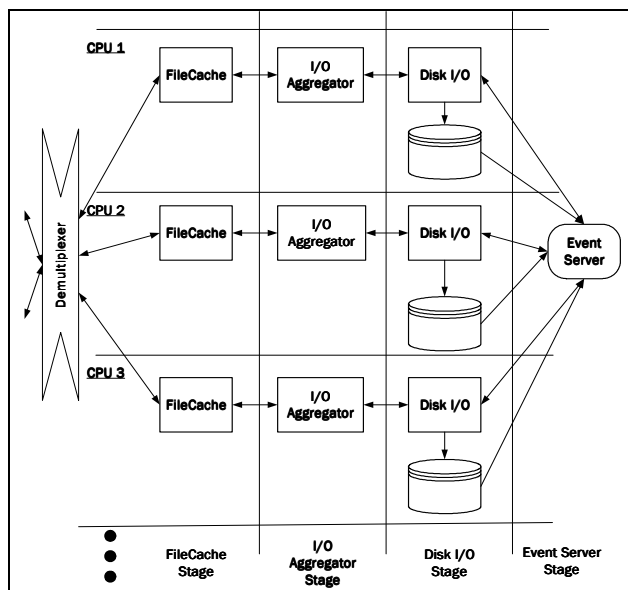

Figure 5. Architecture of staged file cache. Requests for disk blocks are partitioned across processors to avoid sharing the hash table. If a block is not found, it is requested from an I/O aggregator, which combines requests for adjacent blocks and passes them to a disk I/O stage that asynchronously reads the files. When an I/O completes, an event server thread is notified, which passes the completion back to the disk I/O stage.

4 StagedServer Library

The StagedServer library is a collection of C++ classes that implement staged computation and cohort scheduling on either a uniprocessor or multiprocessor. This library enables a programmer to define stages, operations, and policies by writing only application-specific code. Moreover, StagedServer implements an aggressive and efficient version of cohort scheduling. This section briefly describes the library and its primary interfaces.

The library's functionality is partitioned between two principal classes. The first is the *Stage* class, which provides stage-local storage and mechanisms for collecting and scheduling operations. The second is the *Closure* class, which encapsulates an operation and its continuations, provides per-invocation state, and supports invoking an operation and returning its result. The fundamental action in a StagedServer system is to invoke an operation by creating and initializing a closure and handing it to a stage.

4.1 Stage Class

The Stage class is a templated base class that an application uses to derive classes for its various stages. The base class provides the basic functionality for managing closures and for scheduling and executing operations on processors.

4.1.1 Scheduling Policy

StagedServer implements a cohort scheduling policy, with enhancements to increase the processor affinity of data. The assignment of operations to processors occurs when an operation is submitted to a stage. By default, an operation invoked by code running on processor p executes on processor p in subsequent stages. This affinity policy enhances temporal locality and reduces cache traffic, as the operation's data tend to remain in the processor's cache. However, a program can override the default and execute an operation on a different processor when: the processor to execute the operation is explicitly specified, a stage partitions its operations among processors, or a stage uses load balancing to redistribute operations.

A stage maintains a stack and queue for each processor in the system. In general, operations originating on the local processor are pushed on the stack and operations from other processors are enqueued on the queue. When a stage starts processing a cohort, it first empties its stack in LIFO order, before turning to the queue. This scheme has two rationales. Processing the most recently invoked operations first increases the likelihood that an operation's data will reside in the cache. In addition, the stack does not require synchronization, since it is only accessed by one processor, which reduces the common-case cost of invoking an operation.

4.1.2 Processor Scheduling

StagedServer currently uses a simple, wavefront algorithm to supply processors to stages. A programmer specifies an ordering of the stages in an application. In wavefront scheduling, processors independently alternate forward and backward traversals of this list of stages. At each stage, a processor executes operations pending in its stack and queue. When the operations are finished, the processor proceeds to the next stage. If the processor repeatedly finds no work, it sleeps for exponentially increasing periods of time interval. If a processor cannot gain access to an Exclusive stage, because another processor is already working in the stage, the processor skips the stage.

The alternating traversal order in wavefront scheduling corresponds to a common communications pattern, in which a stage passes requests to its successors, which perform a computation and produce a result. It is easy to imagine other scheduling policies, but we have not evaluated them, as this approach works well for the applications we have studied. This topic is worth further investigation.

4.1.3 Thresholds

An orthogonal attribute of a stage is a pair of thresholds that force StagedServer to activate a stage if more than a given number of operations are waiting or after a fixed interval. When either situation arises, StagedServer stops the currently running stage (after it completes its operation), runs the threshold-exceeding stage, and then returns to the suspended stage. For simplicity, an interrupting stage cannot be interrupted, so that other stages that exceed their thresholds are deferred until processing returns to original stage. Thresholds are particularly useful for latency-sensitive stages, such as those interacting with the operating system, which must be regularly supplied with I/O requests to ensure that devices do not go idle.

Another useful refinement is a feedback mechanism, by which a stage informs other stages that it has sufficient tasks. These other stages can suspend processing, effectively turning the processor over to the first stage. So far, voluntary cooperation, rather than hard queue limits, has sufficed.

4.1.4 Partitioned Data

A partitioned stage typically divides its data, so that the operations running on a processor access only a non-shared portion. Avoiding sharing eliminates the need to synchronize access to the data and reduces the cache traffic that results when data are accessed from more than one processor. The current system partitions a variable—using the well-known technique of privatization [30]—by storing its values in a vector with an entry for each processor. Code uses the processor id to index this vector and obtain a private value.

4.2 Closure Class

Closure is a templated base class for defining closures, which are a combination of code and data. StagedServer uses closures to implement operations and their continuations. When an operation is first invoked on a stage, the invoker creates a closure and initializes it with parameter values. Later, the stage executes the operation by invoking one of the closure's methods, as specified by the operation invocation. This method is an ordinary C++ method. When it returns, the method must state whether the operation is complete (and optionally returns a value), if it is waiting for a child to finish, or if it is waiting for another operation to resume its execution.

An operation can invoke operations on other stages—its children. The original operation waits for its children by providing a continuation routine that the system runs when the children finish. This continuation routine is simply another method in the original closure.

The closure passes arguments between a parent and its continuation and results between a child and its parent. This process may repeat multiple times, with each continuation taking on the role of a parent. In other words, these closures are actually multiple-entry closures, with an entry for the original operation invocation and entries for subsequent continuations. In practice, a stage treats these methods identically and does not distinguish between an operation and its continuation.

5 Experimental Evaluation

To evaluate the benefits of cohort scheduling and the StagedServer library, we built two prototypical server applications. The first—a web server—is I/O-bound, as its task consists of responding to HTTP GET requests by retrieving files from a disk or file cache and sending them over a network. The second—a publish-subscribe server—is compute bound, as the amount of data transferred is relatively small, but the computation to match an event against a database of subscriptions is expensive and memory-intensive.

5.1 I/O-Intensive Server

To compare threads against stages, we implemented two web servers. The first is structured using a thread pool (THWS) and the second uses StagedServer (SSWS). We took care to make the two servers efficient and comparable and to share common code. In particular, both servers use Microsoft Window's asynchronous I/O operations. The threaded server was organized in a conventional manner as a thread accepting connections and passing them to a pool of 256 worker threads, each of which performs the server's full functionality: parsing a request, reading a file, and transmitting its contents. This server used the kernel's file cache. The SSWS server also can process up to 256 simultaneously requests. It was organized as a control logic stage, a network I/O stage, and the disk I/O and caching stages described in Section 3.6. The parameters were chosen by experimentation and yielded robust performance for the benchmark and hardware configuration.

As a baseline for comparison, we also ran the experiments on Microsoft's IIS web server, which is a highly tuned commercial product. IIS performed better than the other servers, but the difference was small, which partially validates their implementations.

Our test configuration consisted of a server and three clients. The server was Compaq Proliant DL580R containing four 700MHz Pentium III-Xeon processors (2MB L2 cache) and 4GB of RAM. It had eight 10000RPM SCSI3 disks, connected to a Compaq Smart Array controller. The clients ran on Dell PowerEdge 6350s, each containing four 400MHz Pentium II Xeon processors with 1GB of RAM. The clients and server were connected by a dedicated Gigabit Ethernet network and both ran Windows 2000 Server (SP1).

We used the SURGE benchmark, which retrieves web pages, whose size, distributions, and reference pattern are modeled on actual systems [8]. SURGE measures the ability of a web server to process HTTP GET requests, retrieve pages from a disk, and send them back to a client. This benchmark does not attempt to capture the full behavior of a web server, which must handle other types of HTTP requests, execute dynamic content, perform server management, and log data. To increase the load, we run a large configuration, with a web site of 1,000,000 pages (20.1GB) and a reference stream containing 6,638,449 requests. A SURGE workload is characterized by User-Equivalents (UEs), each of which models one user accessing the web site. We found that we could run up to 2000 UEs per client. All tests were run with the UE workload balanced across the client machines. The reported numbers are for 15 minutes of client execution, starting with a freshly initialized server.

Figure 6 shows the bandwidth and latency of the thread (THWS) and StagedServer (SSWS) servers, and compares them against a commercial web server (IIS). The first chart contains the number of pages retrieved by the clients per second (since requests follow a fixed sequence, the number of pages is a measure of bandwidth) and the second chart contains the average latency, perceived by a client, to retrieve a page.

Several trends are notable. Under light load, SSWS's performance is approximately 6% lower than THWS, but as the load increases, SSWS responds to as many as 13% more requests per unit time. The second chart, in part, explains this difference. SSWS's latency is higher than THWS's latency under light load (by a factor of almost 20), but as the load increases, SSWS's latency grows only 2.3 times, but THWS's latency increases 45 times, to a level equal to SSWS's.

The commercial server, Microsoft's IIS, outperformed SSWS by 4–9% and THWS by 0–22%. Its latency under heavy load was up to 45% better than the other servers' latency.
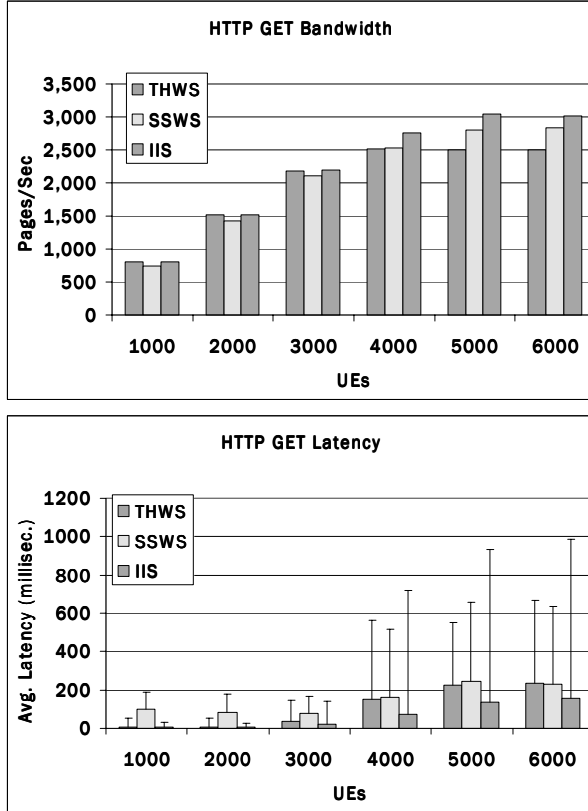
Figure 6. Performance of web servers. These charts show the performance of the threaded server (THWS), StagedServer server (SSWS), and Microsoft's IIS server (IIS). The first records the number of web pages received by the clients per second. The second records the average latency, as perceived by the client, to retrieve a page. The error bars are the standard deviation of the latency.

Figure 7. Processor performance of servers. These charts show the processor performance of the threaded (THWS) and Staged-Server (SSWS) web server. The first chart shows the cycles per instruction (CPI) and the second shows the rate of L2 cache misses.

SSWS performance, which is more stable and predictable under heavy load than the threaded server, is appropriate for servers, in which performance challenges arise as offered load increases. SSWS server's overall performance was relatively better and its processor performance degraded less under load than the THWS server. The improved processor performance was reflected in a measurably improved throughput under load.

5.2 Compute-Bound Server

To evaluate the performance of StagedServer on a compute-bound application, we also built a simple publish-subscribe server. The server used an efficient, cache-friendly algorithm to match events against an in-core database of subscriptions [16]. A subscription is a conjunction of terms comparing variables against integer. An event is a set of assignments of values to variables. An event matches a subscription if all of its terms are satisfied by the value assignments in the event.

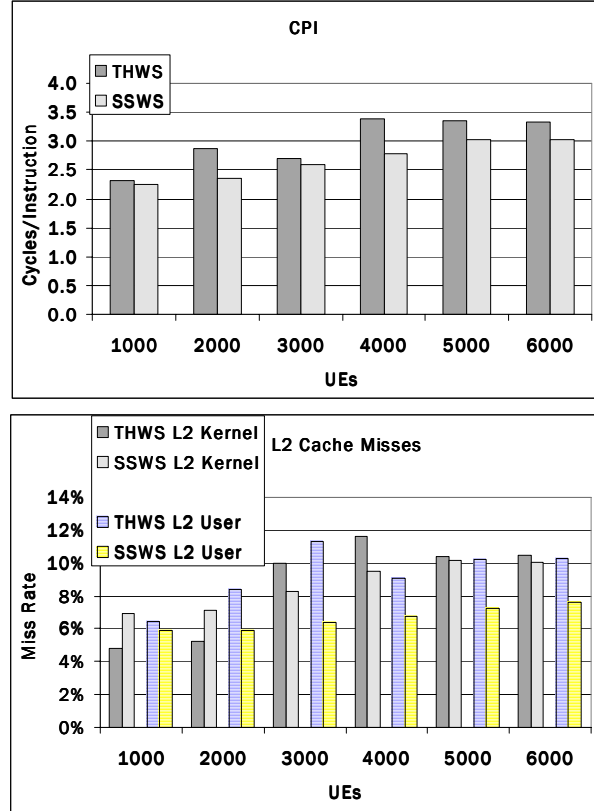Both the threaded (THPS) and StagedServer (SSPS) version of this application shared a common publish-subscribe implementation; the only difference between them was the use of threads or stages to structure the computation. The benchmark was the Fabret workload: 1,000,000 subscriptions and 100,000 events. The platform was the same as above.

The response time of the StagedServer version to events was better under load (Figure 8). With four or more clients publishing events, the THPS responded in an average of 0.57 ms to each request. With four clients, SSPS responded in an average time of 0.53 ms, and its response improved to 0.47 ms with 16 or more clients (21% improvement over the threaded version).

In large measure, this improvement is due to improved processor usage (Figure 8). With 16 clients, SSPS averaged 2.0 cycles per instruction (CPI) over the entire benchmark, while THPS averaged 2.7 CPI (26% reduction). Over the compute-intensive event matching portion, SSPS averaged 1.7 CPI, while THPS averaged 2.5 CPI (33% reduction). In large measure, this improvement is attributable to a greater than 50% reduction in L2 caches misses, from 58% of user-space L2 cache requests (THPS) to a 26% of references (SSPS).

Average Response Time
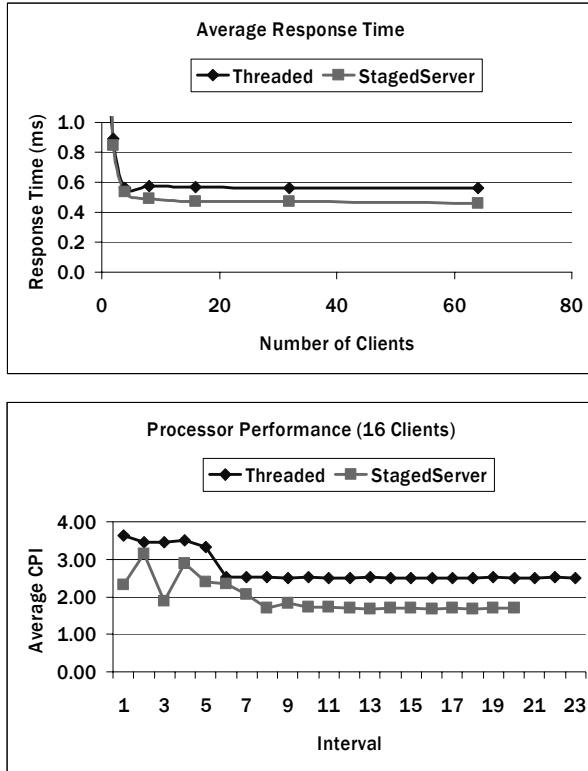

Processor Performance (16 Clients)

Figure 8, Performance of Publish-Subscribe server. The top chart records the average response time to match publish events against subscriptions. The bottom chart compares average cycles per instruction (CPI) of the thread and StagedServer versions over 25 second intervals. The initial part of each curve is the construction of internal data structures, while the flat part of the curves is the event matching.

This application references a large data structure (approximately 66.7MB for the benchmark). When matching an event against subscriptions, Fabret's algorithm, although cache-efficient, may access a large amount of data, and the particular locations are data dependent. StagedServer's performance advantage results from two factors. First, its code is organized so that only one processor references a given subset of the subscriptions, which reduces the number of distinct locations a processor references, and hence increases the possibility of data reuse. Without this locality optimization, SSPS runs at the same speed as THPS. Second, StagedServer batches cohorts of event matches in this data structure. We measured the benefit of cohort scheduling by limiting cohort size. Cohorts of four items reduced SSPS performance by 21%, ten items reduced performance by 17%, and twenty items reduced performance by 9%.

Both optimizations would be beneficial in threaded code, but the structure of the resulting server would be isomorphic to the StagedServer version, with a thread bound to each processor performing event lookups on a subset of the data structure, and a queue in front of each process to accumulate a cohort.

6 Related Work

The advantages and disadvantages of threads and processes are widely known [5]. More recently, several papers have investigated alternative server architectures. Chankhunthod et al. described the Harvest web cache, which uses an event-driven, reactive architecture to invoke computation at transitions in its state-machine control logic [13]. The system, like StagedServer, uses non-blocking I/O; careful avoidance of page faults; and a non-blocking, non-preemptive scheduling policy [7, 26]. Pai proposed a four-fold characterization of server architectures: multi-process, multi-threaded, single-process event driven, and asymmetric multi-process event driven [26]. These alternatives are orthogonal to the task scheduling policy, and as the discussion in Section 2 illustrates, cohort scheduling could increase their locality. Pai's favored event-driven programming style offers many opportunities for cohort scheduling, since event handlers partition a computation into distinct, easily identifiable subcomputations with clear operation boundaries. On the other hand, ad-hoc event systems offer no obvious way to group handlers that belong in the same cohort or to associate data with operations. Section 3 describes staged computation, a programming model that provides a programmer with control over the computation in a cohort.

Welsh recently described the SEDA system, which is similar to the staged computation model [29]. SEDA, unlike StagedServer, does not use explicit cohort scheduling, but instead uses stages as an architecture for structuring event-driven servers. His performance results are similar for I/O intensive server applications.

Blackwell used blocked layer processing to improve the instruction locality of a TCP/IP stack [10]. He noted that the TCP/IP code was larger than the MIPS R2000 instruction cache, so that when the protocol stack processed a packet completely, no code from the lower protocol layers remained in cache for the next packet. His solution was to process several packets together at each layer. The modified stack had a lower cache miss rate and reduced processing latency. Blackwell related his approach to blocked matrix computations [19], but his focus was instruction locality. Cohort scheduling, whose genesis predates Blackwell, is a more general scheduling policy and system architecture, which is applicable when a computation is not as cleanly partionable as a network stack. Moreover, cohort scheduling improves data, not just instruction, locality and reduces synchronization as well.

A stage is similar in some respects to an object in an object-based language, in that it provides both local state and operations to manipulate it. The two differ because objects are, for the most part, passive and their methods are synchronous—though asynchronous object models exist. Many object-oriented languages, such as Java [17], integrate threads and synchronization, but the active entity remains a thread synchronously run a method on a passive object. By contrast, in staged computation, a stage is asked to perform an operation, but is given the autonomy to decide how and when to actually execute the work. This decoupling of request and response is valuable because it enables a stage to control its concurrency and to adopt an efficient scheduling policy, such as cohort scheduling.

Stages are similar in some respects to Agha's Actors [3]. Both start with a model of asynchronous communication between autonomous entities. Actors have no internal concurrency and do not give entities control over their scheduling, but instead presume a reactive model in which an Actor responds to a message by invoking a computation. Stages, because of the internal concurrency and scheduling autonomy, are better suited to cohort scheduling. Actors are, in turn, an instance of dataflow, a more general computing model [23, 24]. Stages also can be viewed as an instance of dataflow computation.

Cilk is language based on a provably efficient scheduling policy [11]. The language is thread, not object, based, but it shares some characteristics with stages. In both, once started, a computation is not pre-empted. While running, a computation can spawn off other tasks, which return their results by invoking a continuation. However, Cilk's work stealing scheduling policy does not implement cohort scheduling, nor is it under program control. Recent work, however, has improved the data locality of work stealing scheduling algorithms [1].

JAWS is an object-oriented framework for writing web servers [18]. It consists of a collection of design patterns, which can be used to construct servers adapted to a particular operating system by selecting an appropriate concurrency mechanism (processes or threads), creating a thread pool, reducing synchronization, caching files, using scatter-gather I/O, or employing various http and TCP-specific optimizations. StagedServer is a simpler library that provides a programming model that directly enhances program locality and performance.

An earlier version of this work was published as a short, extended abstract [21].

7 Conclusion

Servers are commonly structured as a collection of parallel tasks, each of which executes all the code necessary to process a request. Threads, processes, or event handlers underlie the software architecture of most servers. Unfortunately, this software architecture can interact poorly with modern processors, whose performance depends on mechanisms—caches, TLBs, and branch predictors—that exploit program locality to bridge the increasing processor-memory performance gap. Servers have little inherent locality. A thread typically runs for a short and unpredictable amount of time and is followed by an unrelated thread, with its own working set. Moreover, servers interact frequently with the operating system, which has a large and disruptive working set. The poor processor performance of servers is a natural consequence of their threaded architecture.

As a remedy, we propose cohort scheduling, which increases server locality by consecutively executing related operations from different server requests. Running similar code on a processor increases instruction and data locality, which aids hardware mechanisms, such as cache and branch predictors. Moreover, this architecture naturally issues operating system requests in batches, which reduces the system's disruption.

This paper also describes the staged computation programming model, which supports cohort scheduling by providing an abstraction for grouping related operations and mechanisms through which a program can implement cohort scheduling. This approach has been implemented in the StagedServer library. In a series of tests using a web server and publish-subscribe server, the StagedServer code performed better than threaded code, with a lower level of cache misses and instruction stalls and better performance under heavy load.

Acknowledgements

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe, "The Data Locality of Work Stealing," in Proceedings of the Twelfth ACM Symposium on Parallel Algorithms and Architectures (SPAA). Bar Harbor, ME, July 2000.

[2] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur, "Cooperative Tasking without Manual Stack Management," in Proceedings of the 2002 USENIX Annual Technical Conference. Monterey, CA, June 2002.

[3] Gul A. Agha, ACTORS: A Model of Concurrent Computation in Distributed Systems. Cambridge, MA: MIT Press, 1988.

[4] Anastassia G. Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood, "DBMSs on a Modern Processor: Where Does Time Go?," in Proceedings of 25th International Conference on Very Large Data Bases. Edinburgh, Scotland: Morgan Kaufmann, September 1999, pp. 266-277.

[5] Thomas E. Anderson, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors," IEEE Transactions on Parallel and Distributed Systems, vol. 1, num. 1, pp. 6-16, 1990.

[6] Andrew W. Appel, *Compiling with Continuations*. Cambridge University Press, 1992.

[7] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul, "Better Operating System Features for Faster Network Servers," in Proceedings of the Workshop on Internet Server Performance. Madison, WI, June 1998.

[8] Paul Barford and Mark Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation," in Proceedings of the 1998 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems. Madison, WI, June 1998, pp. 151-160.

[9] Luiz André Barroso, Kourosh Gharachorloo, and Edouard Bugnion, "Memory System Characterization of Commercial Workloads," in Proceedings of the 25th Annual International Symposium on Computer Architecture. Barcelona, Spain, June 1998, pp. 3-14.

[10] Trevor Blackwell, "Speeding up Protocols for Small Messages," in Proceedings of the ACM SIGCOMM '96 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication. Palo Alto, CA, August 1996, pp. 85-95.

[11] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou, "Cilk: An Efficient Multithreaded Runtime System," Journal of Parallel and Distributed Computing, vol. 37, num. 1, pp. 55-69, 1996.

[12] Satish Chandra, Bradley Richards, and James R. Larus, "Teapot: A Domain-Specific Language for Writing Cache Coherence Protocols," IEEE Transactions on Software Engineering, vol. 25, num. 3, pp. 317-333, 1999.

[13] Anawat Chankhunthod, Peter Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell, "A Hierarchical Internet Object Cache," in Proceedings of the USENIX 1996 Annual Technical Conference. San Diego, CA, January 1996.

[14] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems," in Proceedings of the Thirteenth ACM Symposium on Operating System Principles. Pacific Grove, CA, October 1991, pp. 122-136.

[15] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.

[16] Françoise Fabret, H. Arno Jacobsen, François Llirbat, Joao Pereira, Kenneth A. Ross, and Dennis Shasha, "Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems," in Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data and Symposium on Principles of Database Systems. Santa Barbara, CA, May 2001, pp. 115-126.

[17] James Gosling, Bill Joy, and Guy Steele, *The Java Language Specification*: Addison Wesley, 1996.

[18] James Hu and Douglas C. Schmidt, "JAWS: A Framework for High-performance Web Servers," in Domain-Specific Application Frameworks: Frameworks Experience By Industry, M. Fayad and R. Johnson, Eds.: John Wiley & Sons, October 1999.

[19] F. Irigoin and R. Troilet, "Supernode Partitioning," in Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages. San Diego, CA, January 1988, pp. 319-329.

[20] Kimberly Keeton, David A. Patterson, Yong Qiang He, Roger C. Raphael, and Walter E. Baker, "Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads," in Proceedings of the 25th Annual International Symposium on Computer Architecture. Barcelona, Spain, June 1998, pp. 15-26.

[21] James R Larus and Michael Parkes, "Using Cohort Scheduling to Enhance Server Performance (Extended Abstract)," in Proceedings of the Workshop on Optimization of Middleware and Distributed Systems. Snowbird, UT, June 2001, pp. 182-187.

[22] James R. Larus, Sriram K. Rajamani, and Jakob Rehof, "Behavioral Types for Structured Asynchronous Programming," Microsoft Research, Redmond, WA, May 2001.

[23] Edward A. Lee and Thomas M. Parks, "Dataflow Process Networks," Proceedings of the IEEE, vol. 83, num. 5, pp. 773-799, 1995.

[24] Walid A. Najjar, Edward A. Lee, and Guang R. Gao, "Advances in the Dataflow Computation Model," Parallel Computing, vol. 251907-1929, 1999.

[25] Yoshihiro Oyama, Kenjiro Taura, and Akinori Yonezawa, "Executing Parallel Programs with Synchronization Bottlenecks Efficiently," in Proceedings of International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA '99). Sendai, Japan: World Scientific, July 1999, pp. 182-204.

[26] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel, "Flash: An Efficient and Portable Web Server," in Proceedings of the 1999 USENIX Annual Technical Conference. Monterey, CA, June 1999, pp. 199-212.

[27] David A. Patterson and John L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2 ed. Morgan Kaufmann, 1996.

[28] Sharon Perl and Richard L. Sites, "Studies of Windows NT Performance using Dynamic Execution Traces," in Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI). Seattle, WA, October 1997, pp. 169-183.

[29] Matt Welsh, David Culler, and Eric Brewer, "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services," in Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01). Alberta, Canada, October 2001, pp. 230-243.

[30] Michael J. Wolfe, *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1995.