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

A server is commonly organized as a collection of concurrent tasks, each of which runs the server's code to process a request. The concurrency is built on threads, processes, or event-driven code, all of which provide control independent tasks and dynamic scheduling to mitigate high-latency operations such as I/O and communication.

Unfortunately, many of these servers run poorly on modern processors. Measurements show that these systems utilize only a fraction of a processor's potential. In part, this poor performance is attributable to the programs' software architecture, which frequently jumps between unrelated pieces of code, thereby reducing the instruction and data locality that is a prerequisite for hardware mechanisms such as caches, TLB, and branch predictors.

This paper describes *cohort scheduling*, a policy that increases code and data locality by batching the execution of similar operations across server requests. *Staged computation* is a programming model that helps structure programs in a manner conducive to cohort scheduling. The *StagedServer* library provides an efficient implementation of cohort scheduling using this model. Measurements show that cohort scheduling improves server throughput as much as 13% and reduces CPI as much as 18%.

1 Introduction

A server 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. These systems make information and resources available to remote clients and facilitate concurrent access to and sharing of data and resources. Server performance is important, as it determines the latency to access a resource and constrains a server's ability to handle multiple requests. Commercial servers, such as databases, are the focus of considerable research to improve the underlying hardware, algorithms and data structures, and efficiency of server code.

Much of this effort focuses on systems' memory hierarchy, which until recently meant disk accesses, but now includes processor caches. This shift reflects the deleterious effects of the expanding processor-memory gap, which is caused by rapidly increasing processor speed and parallelism. In recent processors, loading a word from memory can cost hundreds of cycles, during which three to four times as many instructions could execute. Processors use numerous mechanisms, such as caches, TLBs, and branch predictors, to alleviate this performance mismatch [1]. These mechanisms exploit an observed property of programs—spatial and temporal reuse of code and data—to keep at hand data that is likely to be quickly reused and to predict future program behavior.

Server software often exhibits less program locality and, consequently, achieves poorer performance, than other software. Many studies found that commercial database systems running on-line transaction processing (OLTP) benchmarks incur very high rates of cache misses and instruction stalls, which reduced processors to a fifth to a tenth of their peak performance [2-4]. Some of these performance problems may be attributable to database systems' code size or behavior [5], but their software architecture is also responsible. In these systems, a process or thread

runs for a short period before invoking a system 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 [3]. On an AlphaServer 4100, TPC-B has 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 has an L2 miss rate of 1.2%, an L3 miss rate of 0.32%, and a CPI of 1.62. Databases are important applications, and recent processors have significantly larger cache sizes to accommodate their poor locality [6].

This paper takes the alternative—and complementary approach—of improving program behavior. It presents a new software architecture that enhances instruction and data locality and increases the performance of server software. The architecture consists of a scheduling policy and a programming model. The policy, *cohort scheduling*, consecutively executes a cohort of similar computations from distinct server requests. The 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 increases program locality and improves hardware performance. *Staged computation*, the programming model, provides a convenient programming abstraction with which a programmer can identify and group related computations and make explicit the dependences that constrain scheduling. Staged computation, moreover, reduces the cost of concurrency and the need for expensive, error-prone synchronization. We implemented this scheduling policy and programming model in a reusable library (*StagedServer*). As compared to a threaded server, the StagedServer version performs better under heavy load, delivering as many as 13% more pages while running at a CPI as much as 18% lower.

The paper is organized as follows. The next section discusses parallelism in servers and how existing servers exploit it. Section 3 introduces cohort scheduling and explains how it can improve program performance. Section 4 describes staged computation. Section 5 describes the StagedServer library. Section 6 contains performance measurements. Section 7 concludes.

2 Server Parallelism and Performance

A server is a parallel program with two possible dimensions of concurrency. Typically, the primary source of parallelism is overlapped processing of distinct requests on the server. Two requests can, in general, execute in either order or even run concurrently, since most requests are independent and servers are rarely constrained to process them in a fixed order such as first-come, first-served. Servers exploit this flexibility by suspending a computation when it blocks on a resource, and then initiating or resuming work on another computation. The second dimension of parallelism is concurrency within the computation for a single request. On a parallel computer, a computation can often be broken into concurrent subcomputations.

2.1 Processes and Threads

Many servers organize their architecture by executing computations from distinct requests in separate processes or threads [7]. For simplicity, we will refer to both control regimes as "threads," since the shared address space distinction is incidental to this discussion. When a server request arrives, its processing is assigned to a thread. Either this thread can be created or, more commonly, taken from a fixed-size pool of threads—a device used to limit the number of running threads and mitigate the cost of thread creation, scheduling, and destruction. A thread runs until it voluntarily relinquishes a processor, executes a blocking system call, executes a blocking synchronization operation, or exceeds its time slice.

The thread model is effective for servers because it permits efficient utilization of expensive resources, such as processors, memory, and disks, by rapidly multiplexing them among computations, to fill gaps arising from contention over shared resources and from high-latency opera-

tions such as I/O. In general, when a thread relinquishes a processor, the thread scheduler chooses the subsequent thread without regard to the processor's state. Most scheduling policies focus on system-wide load balancing, ensuring threads' forward progress, or increasing thread affinity for a processor (which does increase reuse of processor state).

From a processor's perspective, threads can lead to desultory execution, in which disjoint code fragments from an application and the operating system run for short periods before passing control to an unrelated fragment. This behavior reduces the locality of a thread's computations, as code interleaved in the thread's execution causes TLB, cache, and branch table conflicts that evict state needed subsequently. Moreover, this form of scheduling does not make good use of latency reduction hardware, as similar computations rarely run consecutively on a processor; looping behavior that increases locality and processor performance.

Considerable evidence attests to the cache effects of context switching. Mogul and Borg studied the effect of process context switches on cache behavior [8]. They found that cache misses added several thousand cycles of delay to a process context switch and this delay was detectable 400,000 instructions after the context switch. Moreover, they observed that I/O intensive processes, which frequently relinquished a processor, seldom run long enough to achieve steady-state cache behavior. Chen measured the memory behavior of an X11 window server [9]. He found that interprocess cache conflicts accounted for a large fraction of the cache misses, particularly in the operating system. Finally, Rosenblum et al. simulated the behavior of commercial applications on a variety of hardware configurations [10]. For the Sybase DB running a TPC-B-like benchmark, they found a significant number of cache conflicts between the OS and DB and observed that larger and more associative caches did not significantly reduce these conflicts. Less evidence documents the effects of thread switching on cache behavior. Jayasimha and Kumar found that most misses in a commercial DB running a TPC-C-like benchmark occurred because of conflicts between different threads or when a thread reloaded state that another thread evicted from the cache [11].

2.2 Event-Driven Servers

Event-driven servers are a recent architecture alternative to processes and threads. A server of this type is structured as reactive system in which external events, such as network or disk I/O, trigger transitions in a state-machine that controls the processing of a request [7]. Each transition invokes a function to perform the next part of a request's computation. The state machines and event-driven framework offer control independence and dynamic scheduling, often at a lower cost than threads. These systems, however, share locality problems with thread-based servers, as subcomputations execute in isolation, with no attempt to schedule to increase locality.

3 Cohort Scheduling

Cohort scheduling is a scheduling policy for improving program locality in servers. The key insight is to identify similar computations in distinct server requests and executing them consecutively on a processor. Because a server receives a continual stream of requests, it can defer processing until a *cohort* of computations arrive at a similar point and then executed them consecutively (Figure 1). 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 increases locality and hardware performance. The benefits of this policy extend beyond a program's code and data. By batching operating system calls, cohort scheduling improves the locality of system, not just application, code.

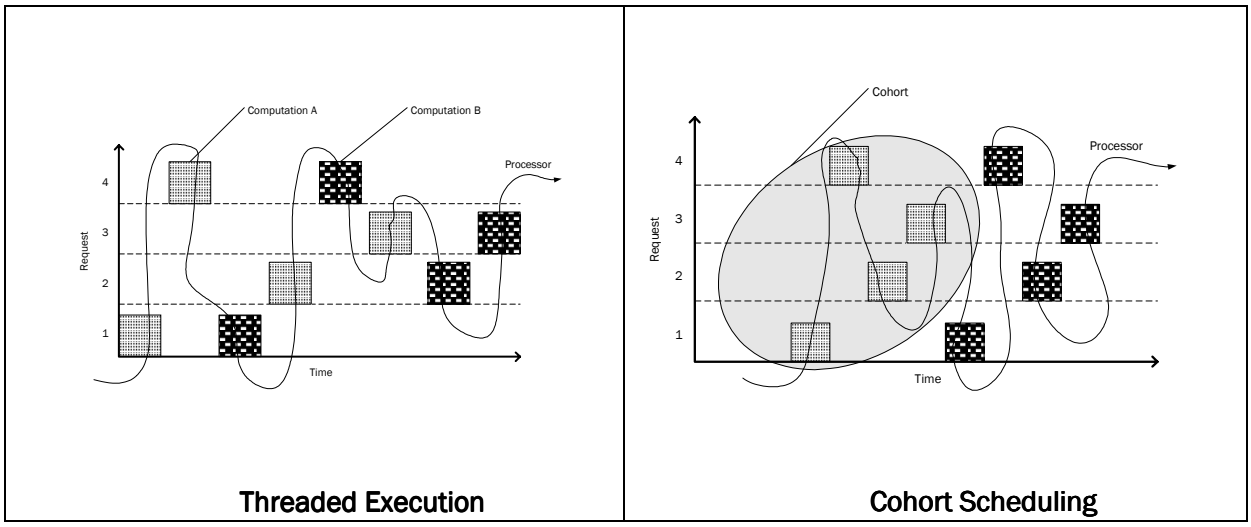

Figure 1. Illustration of cohort scheduling. Shaded boxes indicate computations performed in processing requests, which run along the horizontal axis. Using cohort scheduling, similar computations execute consecutively, which increases program locality and processor performance.

Cohort scheduling exploits two key aspects of server computation: requests are independent and their order of arrival does not constrain their order of completion. Because of these properties, servers can vary tasks' rate of progress and order of completion without affecting correctness. Moreover, servers already possess mechanisms to ensure that the control and data dependencies are preserved despite arbitrary interleavings.

3.1 Cohort Scheduling and Caches

Cohort scheduling would offer little benefit if a processor's cache was arbitrarily large and fully associative, so it could hold all code and data referenced by a server. Except for simple or compute-intensive applications, caches will never achieve this goal. Cohort scheduling increases the opportunity for code and data reuse by avoiding interleaving unrelated code that could cause cache conflicts and evict live cache lines. The principle is similar to loop tiling or blocking, which restructures a matrix computation into submatrix computations that repeatedly reference data before turning to the next submatrix.

A way to evaluate cohort scheduling is to analyze its effect on different types of cache misses, as categorized by Hill's "3C" model [12]. Suppose that cohort scheduling executes computations $C_{i,c}$, $C_{i+1,c}$, …, $C_{n,c}$ from requests i, …, n consecutively on a processor, instead of running them independently in threads. Cohort scheduling reduces cold start misses to the extent that data initially read by computation $C_{k,c}$, $i < k \leq n$ is accessed by an earlier cohort computation, remains in the cache, and would not have been in the cache at the invocation of $C_{k,c}$ in a separate thread. Capacity misses caused by code in $C_{k,c}$, the second "C," are not affected by cohort scheduling. Cohort scheduling, however, can reduce conflict misses, the third "C." Self-conflict, between references in $C_{k,c}$ does not change. However, inter-computation conflicts, between $C_{k,c}$ and $C_{\_,j}$ for $c \neq j$, are eliminated by not running a foreign computation $C_{\_,j}$ in the midst of the $C_{k,c}$'s. This reduction must be balanced against potential conflicts between $C_{k,c}$ and $C_{k+1,c}$. Finally, multiprocessors have a fourth "C," which is the coherence traffic. By aggressively keeping references to a computation's data on a single processor (Section 5.1.1), cohort scheduling reduces coherence misses and the "ping-ponging" that results from data accesses in threads running on different processors.

On the other hand, cohort scheduling can also increase cache misses, if the data used in computation $C_{k,c}$ is evicted from the cache by subsequent members of its cohort, before being reference by its subsequent computation. This effect must be taken into account when forming cohorts, but it can be partially mitigated by the cohort scheduling policy (Section 4.3).

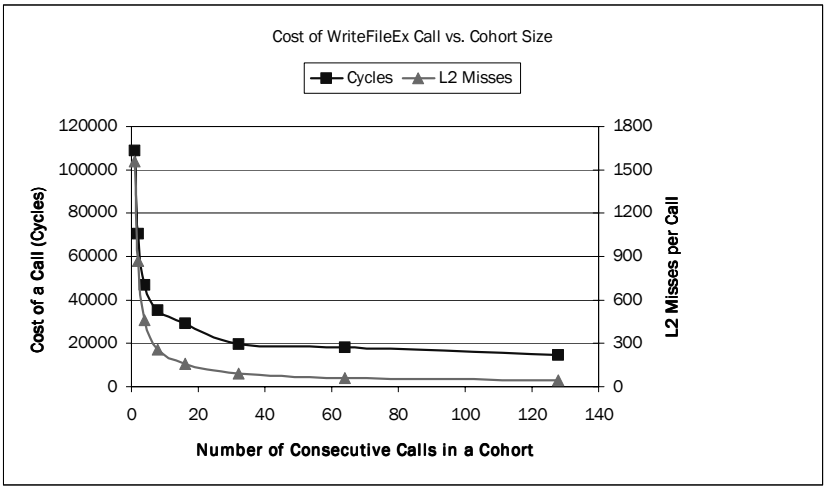

Figure 2. Performance of cohorts of WriteFileEx system calls in Window 2000 Advanced Server (Dell Precision 610 with an Intel Pentium III processor). The chart reports the cost per call—in processor cycles and L2 cache misses—of an asynchronous write to a random 4K block in a file.

Figure 2 illustrates the results of a simple experiment that demonstrates the benefit of cohort scheduling. It reports the cost, per call, of different size cohorts of asynchronous writes to random blocks in a file. Each cohort executed 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 required 109,000 cycles, but the cost dropped to 32% of that value for a cohort of 8 calls and to 16% for a cohort of 64 calls. A direct measure of locality, L2 cache misses, improved more 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.

3.2 Threads and Cohort Scheduling

Cohort scheduling can be used with threads, but the approach forgoes opportunities to identify and form cohorts. The basic idea is simple. A thread scheduler identifies 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 PCs before turning to the next cohort.

The primary advantage of this scheme is simplicity, as it requires only minor changes to the thread scheduler and no changes to applications. The approach, however, has clear shortcomings. In particular, nPC values are an indirect measure of similar program behavior. Only threads with identical nPCs end up in a cohort, which misses many opportunities to identify code with similar behavior. For example, several routines (with different PCs) that access a data structure could form a cohort. Extensions, such as using the distance between PCs as a measure of similarity, have little connection to logical behavior and are perturbed by code scheduling and compiler linking. In addition, the scheme identifies cohorts at the point at which a thread resumes execution, which may not correspond to operation boundaries. For example, programs invoke write system calls at many places. To assembly a cohort, the scheduler would need a wrapper routine, so it could suspend threads before the call. Finally, given the large number of points at which a thread suspends, the expected number of threads in each cohort will be small, which partially defeats cohort scheduling.

3.3 Previous Work

The advantages and disadvantages of threads and processes are widely known [13]. More recently, several papers have investigated alternative architectures for servers. Chankhunthod et al. described the Harvest web cache [14]. It uses an event-driven architecture that, similar to reactive systems, invokes computation at transitions in a state-machine driven by external events such as I/O. The published description briefly alludes to a non-blocking I/O library; careful avoidance of page faults; and a non-blocking, non-preemptive scheduling policy, and other papers elucidate this description [7, 15]. Pai proposed a four-fold characterization of server architectures: multi-process, multi-threaded, single-process event-driven, and asymmetric multi-process event-driven [7]. These alternatives are orthogonal to the scheduling policy, and as the discussion in Section 3.2 illustrates, cohort scheduling could be used to increase their locality. As shown below, an event-driven programming style offers more opportunities to apply this scheduling policy, since the event handlers partition a computation into distinct, easily identifiable subcomputations with clear operation boundaries. On the other hand, event systems offer no obvious way to identify distinct handlers that should be put in the same cohort or ways to associate data with operations. Section 4 describes staged computation, an event-based architecture that provides a programmer with control over the computation in a cohort, while retaining the low overhead.

Blackwell used blocked layer processing to improve the instruction locality of a TCP/IP stack [16]. 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 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 [17], but he changed the focus to 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 partitioned and easily rescheduled as a network stack. Moreover, this paper also shows that cohort scheduling improves data, not just instruction, locality and reduces synchronization as well.

4 Staged Computation

Staged computation is a programming model that replaces processes and threads. It offers compelling performance and correctness advantages and is particularly amenable to cohort scheduling. In this model, a program is built from a collection of *stages*, each of which consists of a set of related operations and data. An operation on a stage is invoked asynchronously, so its invocation, execution, and response are decoupled. A stage has *scheduling autonomy*, which enables it to control the order and concurrency of its operations.

A stage, similar to a class in an object-oriented language, consists of local state and operations. However, stages differ from objects in two major respects. First, a stage operation is invoked asynchronous, so that its sender does not wait for the computation to complete, but instead continues and rendezvouses later, if necessary, to retrieve a result. Second, stages have autonomy to control the execution of its operations. This autonomy extends to deciding when and how to execute the computation associated with an operation.

This programming model facilitates cohort scheduling because a stage provides a natural mechanism to group operations with similar behavior and locality and the autonomy to exploit cohort scheduling. Stages provide additional programming advantages as well. Stages control their internal concurrency, which promotes a programming style that reduces the need for expensive, error-prone explicit synchronization. This section first describes the programming model, then provides an example (Section 4.4), and discusses related work (Section 4.5). Section 5 presents an implementation of the model in a C++ class library.

4.1 Stages

Programmers collect operations in a stage for four reasons. The first is to group computations that share program state and control access to their data. Operations grouped this way are obvious candidates for cohort scheduling, because it enhances the temporal locality of references to this shared data. 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 that collecting them in a stage helps identifies operations that could benefit from cohort scheduling. The third is to encapsulate control logic in the form of a Finite-State Automata (FSA).

The final reason to group operations is to enable the stage to control concurrency, both to reduce interprocessor cache traffic and the need for synchronization. A stage's message-passing-like interface provides a bridge between the possibly high degree of concurrency in the rest of the program and a tightly controlled model within a stage. For example, an exclusive stage executes at most one operation concurrently, regardless of the system parallelism. This policy, which is obviously not suitable for all stages, eliminates the need to synchronize accesses to stage-local data. A stage's control of concurrency enables it to trade simplicity and efficiency against demands for performance, for a limited portion of a program's computation.

4.2 Operations

Operations are asynchronous computations that stages run at their discretion. When an operation executes, it can invoke any number of child operations on any stage, including the one running its parent. Invocation of an operation is unconnected to its execution, so a parent and its children run independently. When the parent finishes, it can wait for its children to finish, retrieve the results of their computation, and invoke a continuation to 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*. After they complete and return their results, *op-a*'s continuation (*op-a-cont*) runs and processes the children's' results.

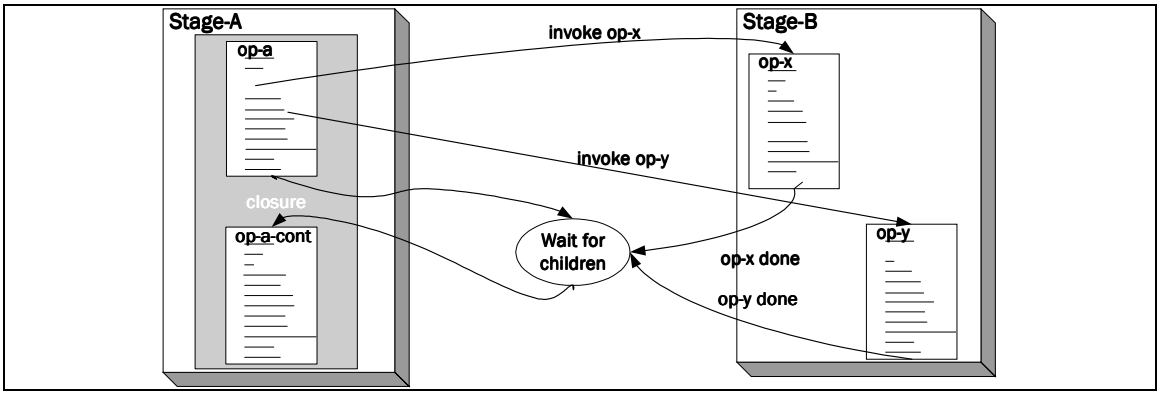

Figure 3. Example of stages and operations. Stage-A runs op-a, which invokes two operations in Stage-B and waiting until they complete before running op-a's continuation.

Code within an operation runs sequential. It can invoke both conventional—synchronous—procedure calls and asynchronous operation. However, once started, an operation runs until it finishes, at which point it relinquishes the processor to the stage. Because operations terminate, a stage need not save processor state and stack, which saves space and time [18]. An operation that suspends and waits for an event—such as I/O, synchronization, or another operation's completion—provides a continuation that runs after the event occurs. Both the original operation and its continuation share *a closure,* a per-invocation structure that stores and pass values between the two.

Asynchronous operations offer low-overhead parallelism, which enables a programmer to exploit fully the concurrency within an application. The overhead, in time and space, of an operation invocation is close to a procedure call. As described in Section 5, invoking an operation entails allocating and initializing a closure and passing it to a stage. Since an operation runs to completion, it does not require its own stack or area to preserve processor state, which eliminates much of the cost of creating a thread. Similarly, returning a value and re-enabling a continuation are inexpensive operations.

Staged computation supports a variety of programming styles, including software pipelining, event-driven state machines, bi-directional pipelines, and fork-join parallelism. Typically, stages in a server are arranged as a pipeline in which requests arrive at one end and response flow out the other. This style of computation is easily programmed by packaging a request as a structure passed between stages. Linear pipelining of this sort is efficient, because a stage retains no information on a completed computation.

A common instance of this style is event-driven reactive programming. Typically, a finite state automata's describing the reactive system is encapsulated within a stage (e.g., Figure 5). Its initial state is specified by an operation that arrives at the stage. The operation's closure holds the state of the FSA (and related computations). Transitions along edges in the FSA occur when children complete or events re-enable a continuation. These computations conceptually occur along edges in the FSA. Each runs until it blocks by specifying the next state in the FSA. Often, a FSA in one stage will communicate with FSAs in other stages, which opens the possibility of applying techniques, such as model checking [19], to verify properties of the entire system.

However, stages are not constrained to a linear style. Another common style is bi-directional pipelining, which is close to a procedure call and return. In this style, a stage passes a subtask to another stage and resumes the computation when the result appears. This style relies on breaking an operation into series of subcomputations, which are run as continuations when results appear. Although programmers currently must partition their computation by hand, a compiler could easily translate code into this style, which is close to continuation passing [20, 21]. Since closures are first-class objects, a stage can pass its closure to an operation and wait to be signaled, which allows a programmer to construct non-linear control structures.

4.3 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 in what order. This scheduling freedom allows several refinements of cohort scheduling that reduce the need for expensive and error-prone synchronization. In particular, we found three policies to be useful:

- An *exclusive stage* executes at most one of its operations on a single processor at a time. Since operations run sequentially and completely, access to stage-local data does not need synchronization. When 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 is amortized over a cohort of operations [22].
- A *partitioned stage* partitions invocations (based on a key passed as a parameter), so that operations on different processors need not share data. For example, consider a file cache stage that partitions requests using a hash function on the file number, so that each processor maintains its own hash table of in-memory disk blocks. 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 any 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 the staged computation framework.

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. The scheduling policy exploits the potential locality and scheduling freedom by consecutively running similar operations on a processor.

4.4 Stage Computation Example

As an example of staged computation, consider the file cache used in the web server described in Section 6. A file cache, an important component of many servers, stores recently accessed disk blocks in memory. It maps block identifiers to copies of disk blocks. Its primary data structure is a hash table, which, in a threaded program must be protected with mutexes.

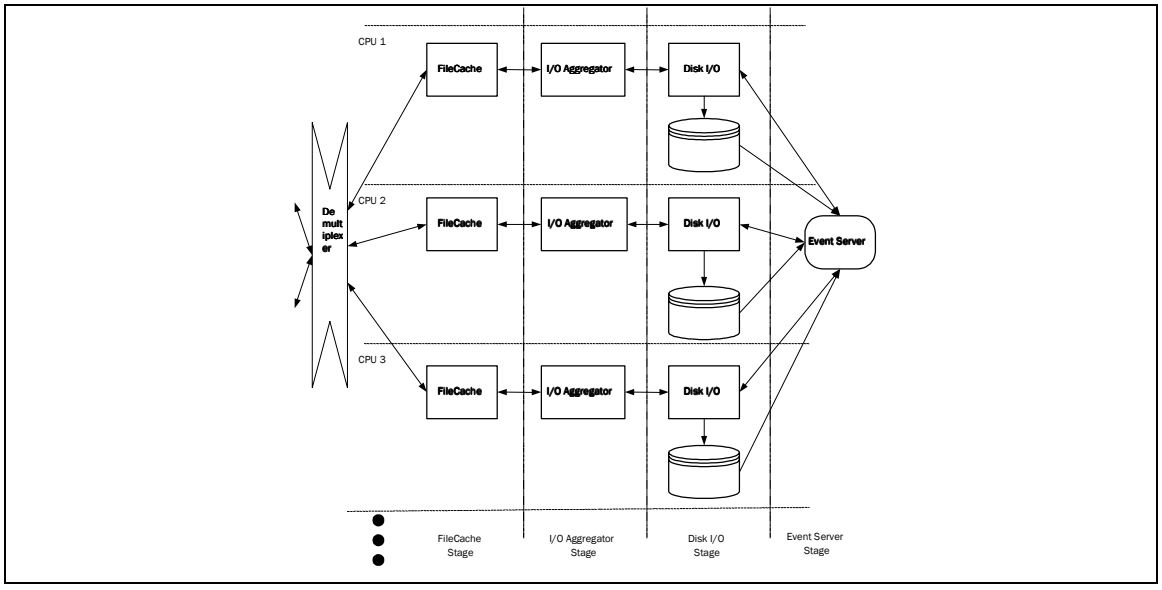

Figure 4. Architecture of staged file cache. Requests for disk blocks are partitioned across CPUs to avoid sharing the hash table. If a block is not found, it is requested from an I/O aggregator, which combines requests for adjacent blocks and passes them to a disk I/O stage that asynchronously reads the files. When the I/O finishes, an event server is notified, which passes this information back to the disk I/O stage.

The staged file cache consists of three partitioned stages (Figure 4). The cache is logically partitioned across the processors, so that each CPU manages a unique subset of the files. This partition uses a hash of a file identify as a key (for large files, their disk blocks could be striped across the table). The stage maintains a hash table per processors, which map file identifiers to memory-resident disk blocks. Since each table is only referenced by a single processor, the hashing code needs 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 for a cohort, and combining them at the end of a cohort's processing before passing the larger I/O request to 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 the pending I/O. This operation registers itself and then suspends until the I/O completes. This stage interfaces with the operating system's asynchronous notification mechanism. It runs in separate threads that wait on an I/O Completion Port, which the system uses to signal completion of asynchronous I/O. Upon notification, this stage matches an event with a waiting closure, which it re-enables, passing along information from the Completion Port. The Disk I/O stage in turn returns the disk blocks to the I/O Aggregator, which passes them to the FileCache stage, where it is recorded and passed back to the client.

4.5 Previous Work

A stage is similar to an object in object-oriented languages, in that both provide local state and operations to manipulate it. The two concepts differ in that objects in most models are passive and their methods are generally synchronous—though asynchronous object models exist. Programming languages, such as Java [23], provide mechanisms for controlling concurrency among method invocations, but the active entity remains the outside agent that invokes methods 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 [24]. 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 respond 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 [25, 26]. Stages also can be viewed as an instance of dataflow computation.

Cilk is language based on a provably efficient scheduling policy [27]. The language is thread, not object, based, but it shares some characteristics with stages. In both, once started, a computation is not preempted. 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 [28].

5 StagedServer Library

The StagedServer library is a collection of C++ classes that implement staged computation and cohort scheduling on either a uniprocessor or a symmetric multiprocessor. This library enables a programmer build stages, operations, and policies by defining only application-specific code. Moreover, StagedServer implements an aggressive and efficient form 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 state 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.

5.1 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 operation on processors. The STAGE class constructor takes the following arguments:

```
STAGE::STAGE(const char *Name,
             STAGE_TYPE Type,
             bool BalanceLoad = false,
             int CacheSize = 0,
             int BatchThreshold = 0,
             int BatchTimer = DefaultTimer,
             bool MaintainOrder = false,
             int MaxBatchSize = StageBatchSize)
```

The first two arguments name the stage and specify its type, which currently is one of: ExclusiveStage, PartitionedStage, or SharedStage (Section 4.3).

The BalanceLoad flag causes a stage to attempt to equalize the quantity of operations assigned to each processor. When this flag is set and a processor finds its quantity of operations below a threshold (currently 4), the processor steals closures from other processors' queues, to obtain *1/P* of the waiting operations. This option is not applicable in partitioned stages, which divide work by semantic rules. This option sets the MaintainOrder flag.

Load balancing can utilize a SMP more effectively, but it incurs significant costs. Determining another processor's load and moving operations is reasonably inexpensive, but shifting tasks may decrease data locality. However, balancing an early stage can have a considerable downstream effect and help keep the system balanced.

CacheSize, if non-zero, provides an Exclusive stage with an estimate of the size of an operation's shared state, which it uses to decide whether to execute a cohort of computations on the current processor, or leave them for subsequent processing on their home processor.

The BatchThreshold and BatchTimer values, if non-zero, force StagedServer to activate a stage either when too many operations are waiting or after a fixed interval. For either one, StagedServer stops the current stage (after it completes its operation), runs the threshold-exceeding stage, and then returns to the suspended stage. 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.

The MaintainOrder flag forces the stage to process operations in the order in which they arrive. If the flag is false, a stage uses a locality-aware policy described below.

Finally, MaxBatchSize specifies the maximum number of operations processed in a cohort. If left unspecified, the stage processes all outstanding operations, including those that arrive during this processing.

5.1.1 Scheduling Policy

StagedServer implements a cohort scheduling policy, with enhancements to increase the processor affinity of data. In general, when a stage activates a processor, it consecutively executes all accumulated operations. The parameters described above modify this behavior slightly. A preliminary assignment of operations to processors is done when an operation is submitted to a stage. In general, an operation invoked by code running on processor *p* will execute on processor *p* in subsequent stages. This affinity policy enhances temporal locality and reduces cache traffic, as the operation's data tends to remain in the processor's cache. However, an operation can move to another processor when: the program specifies the processor to execute the operation, when a stage partitions the operations among processors, or when 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 a stack and operations from other processors are enqueued on the queue. If the MaintainOrder is set, all operations are enqueued. 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. First, processing the most recently invoked operations first increases the likelihood that an operation's data will be in the cache when its code runs. Second, the stack does not require synchronization, since it is only accessed by one processor, which reduces the common-case cost of invoking an operation.

5.1.2 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. Processors independently alternate forward and backward traversals of this list. At each stage, a processor executes operations pending its stack and queue. When they are empty, the processor proceeds to the next stage. If the processor repeatedly finds no work, it sleeps for an increasing amount of time, using an exponential backoff 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 corresponds to a common communications pattern, in which a stage passes a request to its successors, which perform a computation and produces a result. The wavefront order is mitigated by the two thresholds, as discussed above. 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.

5.1.3 Partitioned Data

A partitioned stage typically divides a data structure, so that the operations running on a processor access only a non-shared piece of the structure. Avoiding sharing eliminates the need to synchronize access to the data and reduces the cache traffic that results when data is accessed from more than one processor. The current system partitions a variable—using the well-known technique of privatization [29]—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.

5.2 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 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. It waits for a child to produce a result by providing a continuation that runs when the children finish. This continuation 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 continuation.

Consider, for example, the code fragment from a web server in Figure 5. EstablishConnection is the operation in the WebStage that creates a new connection. This operation is a method in WEB_CLOSURE. When the stage executes this method, it tells the

NetworkStage that it is waiting for an incoming connection and passes a structure (in its closure) in which to return connection information. The method finishes waiting for its child and by naming its continuation (ReadRequest). When the connection is established, the NetworkStage re-enables the closure. The WebStage subsequently executes this closure, which runs the ReadRequest method. This code normally passes a read request to the network stage and waits for a response to pass to the HTTP parser. If the connection fails, the method directly calls EstablishConnection, which again tries to get a connection.

```
ACTIONS WEB_CLOSURE::EstablishConnection() {
   NetworkStage->CreateIncomingConnection(&NWCreateResult);

   return WaitForChildren(ReadRequest);
}

ACTIONS WEB_CLOSURE::ReadRequest() {
   if (0 == NWCreateResult->LastError) {
    info->ConnectionNumber = NWCreateResult->ConnectionNumber;
    NetworkStage->ReadFromConnection(&NWReadWriteResult, info->ConnectionNumber,
                                          info->StrBuffer, sizeof(info->StrBuffer));

    return WaitForChildren(ParseRequest);
   }
   else {
    return EstablishConnection();
   }
}
```

Figure 5. Example of StagedServer code. This code is part of a web server. It waits for an incoming connection request, then reads the HTTP command and sends it to a parser.

5.2.1 Creating Closures

StagedServer combines creation and initialization of a closure by overload the new operation in the CLOSURE class. In C++, the new operator can take a set of parameters distinct from those passed to an object's constructor. StagedServer uses the new operator's parameters to specify the stage that will execute the operation. For example, the first operation in Figure 5, calls the following method in the NetworkStage:

```
NETWORK_STAGE::CreateIncomingConnection(RESULT<CREATE_RESULT> *Result) {
      NETWORK_CLOSURE* x =
          new(NETWORK_CLOSURE::CreateIncomingConnection, this, Result, MaxProcNumber)
          NETWORK_CLOSURE( );
      x->Start( );
   }
```

The new operator's first parameter list specifies the member function to start the computation (CreateIncomingConnection), the stage that executes the operation (this), a location for the operation's result, and a partition key. The second parameter list, to NETWORK_CLOSURE's constructor, in this case does not pass any arguments.

CLOSURE's parameterized new function takes the following arguments:

```
VOID *CLOSURE::new(unsigned int Size,
                   FUNCTION Function,
                   STAGE *TargetStage,
                   VALUE *ResultLoc = NULL,
                   int PartitionKey = NoPartitionKey,
                   int SessionKey = NoSessionKey)
```

Size is the size of the closure. Function is a method in the closure that is invoked to execute the operation. The method has a result type of ACTION and no arguments. Instead, its parameters are passed to the closure's constructor, which stores them. The closure's methods reference these values as class instance variables, which appear syntactically similar to formal parameters.

TargetStage is the stage that will run the operation. ResultLoc is the location where the operation should return its result. PartitionKey is the value used to distribute work among processors in a partitioned stage. In other types of stages, PartitionKey specifies which processor should execute the operation. SessionKey names a group of computations, across one or more stages, which are collectively controlled (i.e. terminated).

When an operation, either the original or a continuation, finishes execution, it must inform its stage of its state by calling one of the following functions:

```
ACTION CLOSURE::Complete(VALUE *Value = NULL)
ACTION CLOSURE::DispatchChildren(FUNCTION Continuation = NoFunctionCall)
ACTION CLOSURE::WaitForChildren(FUNCTION Continuation = NoFunctionCall)
ACTION CLOSURE::WaitForEvent(FUNCTION Continuation = NoFunctionCall)
```

The Complete function specifies that an operation is finished and provides a value to return. This function also indicates that the operation's children need not return results, as no continuation will see them, and that the parent closure should be deallocated. DispatchChildren starts the children, but does not wait for them, as it immediately invokes the specified continuation. WaitForChildren starts the children and waits for them to complete before invoking the specified continuation. WaitForEvent starts the children and suspends the current operation. The continuation can be re-enabled later by passing its closure to STAGE::StartServerOperation.

When an operation is running, it can invoke other operations. However, StagedServer does not pass these operations to their stages immediately, because before the parent terminates, the parent's stage does not know the total number of children or the parent's continuation. Deferring the passing of child closures to their stages enables the system to optimize the rendezvous between parent and child. If the parent is not going to wait for its children, they are marked as orphans, which means that they cannot return a result and need not inform their parent when they finished. If the parent has only one child, it does not require synchronization and may directly re-enable the parent's continuation. If there are multiple children, they decrement a counting semaphore, initialized with the number of children, and the last one re-enables the parent's continuation. Of course, deferring prevents execution overlap between a parent and its children, which simplifies the programming model, by eliminating races between parent and child. In practice, the lack of overlap has little effect, as a parent frequently suspends immediately after starting its children and because children run while their parent's stage executes other operations.

5.3 Related Work

JAWS is an object-oriented framework for writing web servers [30]. 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.

6 Experimental Evaluation

To evaluate the benefits of the StagedServer library, we chose the prototypical server application of delivering static web pages, which consists of reading a small request string from the network, translating it into a file name, and sending the page to the client. Because this application is simple and demanding, it was possible to carefully tune the code. Unfortunately, this type of server spends most of its time in the operating system, so the range of performance improvement is not large.

6.1 Experiment Setup

We implemented two web servers. The first version used threads (TH), the second used StagedServer (SS). We took care to make the two servers efficient and comparable and to share

common code. In particular, both versions use asynchronous I/O operations. The threaded server was structured in a conventional manner as a single thread accepting connections and passing them to a pool of 256 worker threads, which parse a request, read and transmitted a file, and then get another connection. The SS server also processed up to 256 requests simultaneously. It was organized as described above. The parameters were chosen by experimentation and yielded robust performance for the benchmark and hardware configuration.

Our test configuration consisted of a server and three clients[1]. The server was Compaq Proliant DL580R, which contained four 700MHz Pentium III-Xeon processors (2MB L2 cache) with 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. Both server and clients ran Microsoft Windows 2000 Server (SP1).

As a test, we used the SURGE benchmark, which uses a collection of clients to retrieve a variety of web pages whose size, distributions, and reference pattern are modeled on actual systems [31]. SURGE measures the ability of a web server to process HTTP GET requests, retrieve pages from a disk, and push them through the network. 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 the number of User-Equivalents (UEs), each of which models one user accessing the web site. We found we could run up to 2000 UEs per client. All tests were run with the total UE workload balanced across the client machines. The reported numbers are for 15 minutes of client execution, starting with a freshly initialized server.

6.2 Measurements

Figure 6 records the bandwidth and latency of the thread (TH) and StagedServer (SS) servers, and compares them against a commercial web server. The first chart shows the number of pages retrieved by the clients in the test interval (since requests follow a fixed sequence, the number of pages is a measure of bandwidth) and the second chart shows the average latency to get a pages. Several trends are notable. Under light load, SS responds to approximately 6% fewer requests than TH, but as the load increases, SS responds to as many as 13% more requests. The second chart, in part, explains this change. SS's latency is far higher than TH's latency under light load (by a factor of almost 20), but as the load increases, SS's latency grows only 2.3 times, but TH's latency increases 45 times, to a level equal to SS's latency.

The commercial server, Microsoft's IIS, outperformed SS by 4–9% and TH by 0–22%. Its latency under heavy load was up 45% better than the other servers' latency. Although IIS is a far more general web server, it provides a useful baseline for comparison. In particular, IIS handles static pages highly efficiently by using the Windows TransmitFile system call to copy a file to a network connection in a single system call.

Figure 7 contains several processor performance metrics collected while running the TH and SS servers. The first is the average cycles per instruction (CPI), which is 3–18% lower for SS than TH. The next metric is the fraction of cycles in which the processor encountered a stall of some sort. Note that with the Pentium's dynamic instruction scheduling, computation typically continues during these cycles. Under light load, SS incurred 5–35% fewer of these cycles, but the gap closed as the load increased. The other two charts show the rate of L2 (unified) and L1 data (DCU) cache misses, broken down by kernel and user code. As the load increased, SS's L2 kernel

[1] We need to rerun these experiments with a fourth client, as three only drove the server to 50–60% average (80–90% peak) load.

misses increased from 6.9% to 10.2%, but TH's misses increased from 4.8% to as much as 11.6%. The DCU misses in the kernel showed a similar pattern, with SS incurring slightly more misses under light load and fewer misses under heavy load. The user-level cache misses show that SS, despite having a far larger and more complex run-time system (StagedServer), incurred significantly fewer user-level L2 misses at all loads and nearly a constant rate of user-level DCU misses.

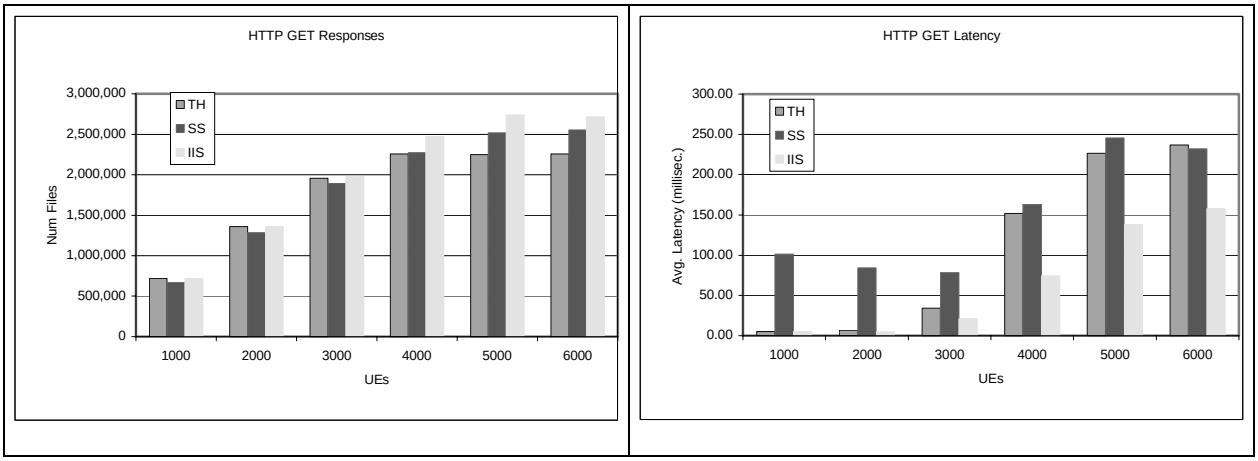

Figure 6. Performance of web servers. These charts show the performance of the threaded server (TH), StagedServer server (SS), and Microsoft's IIS server (IIS). The first records the number of files transmitted by each server. The other records the client's average latency to retrieve a file.

SS performance, which is more favorable under heavy load, is good for servers, in which the performance challenges arise as the demands increase. SS server's overall performance was relatively better and its processor performance degraded less under load than did TH server. Its user-level cache miss rates were consistently lower than the threaded code and did not vary much under load. Moreover, its kernel miss rate increased only 46%, as compared to a 141% increase in the threaded code's miss rate.

7 Conclusion

Servers are commonly structured as a collection of parallel tasks, each of which runs the server's code to process a request. Threads, processes, or event handlers underlie the software architecture of many servers. Unfortunately, these systems 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 task, with its own working set. Moreover, servers interact frequently with the operating system, which has a large and disruptive working set. This paper argues that poor processor performance of servers is a natural consequence of their software architecture.

As a remedy, we propose cohort scheduling, which increases server locality by consecutively executing logically 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 sample web server, the StagedServer code

performed better than threaded code, with a lower level of cache misses and instruction stalls and better server performance under heavy load.

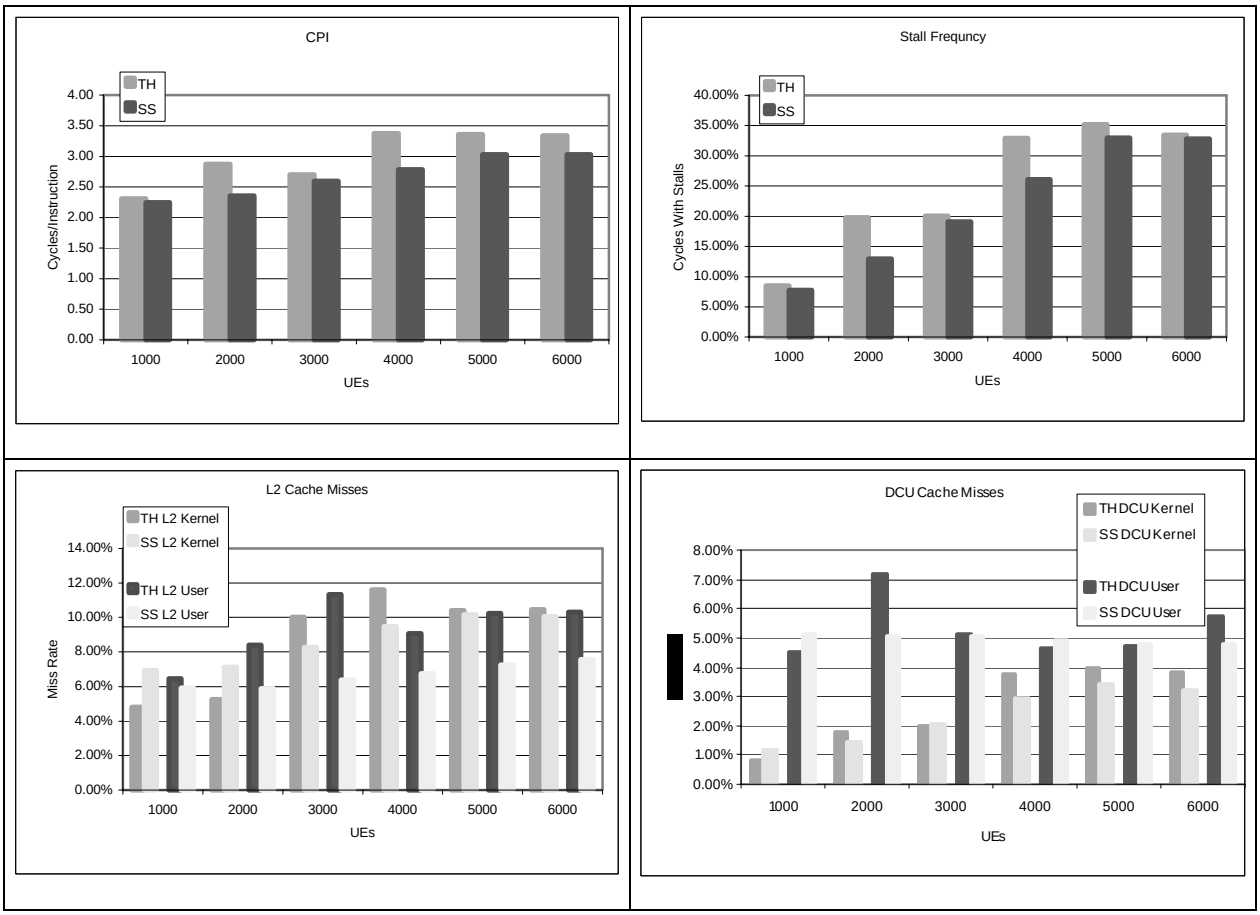

Figure 7. Processor performance of servers. These charts show the processor performance of threaded (TH) and StagedServer (SS) server. The first chart shows the cycles per instruction (CPI), the second show the fraction of cycles in which the processor encountered a stall of some type, the third show the rate of L2 cache misses, and the final one show the rate of L1 data cache (DCU) misses.

Acknowledgements

This work has on going for a long time, and countless people have provided invaluable insights and feedback. This list is incomplete; and we apologize in advance if your name should be here. Rick Vicik made important contributions to the idea of cohort scheduling and the early implementations. Jim Gray has been a ceaseless supporter and advocate of this work. Kevin Zatloukal helped write the web server and run many early experiments. Trishul Chilimbi, Jim Gray, Vinod Grover, Mark Hill, Murali Krishnan, Paul Larson, Milo Martin, Ron Murray, Luke McDowell, Scott McFarling, Simon Peyton-Jones, Mike Smith, and Ben Zorn provided many helpful questions and comments. Jay Jayasimha went out of his way to explain his fascinating, but little-known measurements. Alessandro Forin, Galen Hunt, and Trishul Chilimbi provided machines for the experiments.

References

[1] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach*, 2 ed: Morgan Kaufmann, 1996.

[2] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. 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, 1998, pp. 15-26.

[3] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*. Barcelona, Spain, 1998, pp. 3-14.

[4] A. G. Ailamaki, D. J. DeWitt, M. D. Hill, and D. 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, 1999, pp. 266-277.

[5] S. Perl and R. 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, 1997, pp. 169-183.

[6] Z. Cvetanovic and R. E. Kessler, "Performance Analysis of the Alpha 21264-based Compaq ES40 System," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*. Vancouver, Canada, 2000, pp. 192-202.

[7] V. S. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An Efficient and Portable Web Server," in *Proceedings of the 1999 USENIX Annual Technical Conference*. Monterey, CA, 1999, pp. 199-212.

[8] J. C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Santa Clara, CA, 1991, pp. 75-85.

[9] J. B. Chen, "Memory Behavior of an X11 Window System," in *Proceedings the 1994 USENIX Winter Technical Conference*, 1994, pp. 189-200.

[10] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta, "The Impact of Architectural Trends on Operating System Performance," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*. Copper Mountaint Resort, CO, 1995, pp. 285-298.

[11] J. Jayasimha and A. Kumar, "Thread-based Cache Analysis of a Modified TPC-C Workload," in *Proceedings of the Second Workshop on Computer Architecture Evaluation Using Commercial Workloads*. Orlando, FL, 1999.

[12] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE Transactions on Computers*, vol. C-38, pp. 1612-1630, 1989.

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

[14] A. Chankhunthod, P. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, "A Hierarchical Internet Object Cache," in *Proceedings of the USENIX 1996 Annual Technical Conference*. San Diego, CA, 1996.

[15] G. Banga, P. Druschel, and J. C. Mogul, "Better Operating System Features for Faster Network Servers," in *Proceedings of the Workshop on Internet Server Performance*. Madison, WI, 1998.

[16] T. 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, 1996, pp. 85-95.

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

[18] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. 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, 1991, pp. 122-136.

[19] J. Edmund M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA: MIT Press, 1999.

[20] A. W. Appel, *Compiling with Continuations*: Cambridge University Press, 1992.

[21] S. Chandra, B. Richards, and J. R. Larus, "Teapot: A Domain-Specific Language for Writing Cache Coherence Protocols," *IEEE Transactions on Software Engineering*, vol. 25, pp. 317-333, 1999.

[22] Y. Oyama, K. Taura, and A. 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, 1999, pp. 182-204.

[23] J. Gosling, B. Joy, and G. Steele, *The Java Language Specification*: Addison Wesley, 1996.

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

[25] E. A. Lee and T. M. Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, vol. 83, pp. 773-799, 1995.

[26] W. A. Najjar, E. A. Lee, and G. R. Gao, "Advances in the Dataflow Computation Model," *Parallel Computing*, vol. 25, pp. 1907-1929, 1999.

[27] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Journal of Parallel and Distributed Computing*, vol. 37, pp. 55-69, 1996.

[28] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The Data Locality of Work Stealing," in *Proceedings of the Twelfth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*. Bar Harbor, ME, 2000.

[29] M. J. Wolfe, *High Performance Compilers for Parallel Computing*: Addison-Wesley, 1995.

[30] J. Hu and D. 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, 1999.

[31] P. Barford and M. 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, 1998, pp. 151-160.